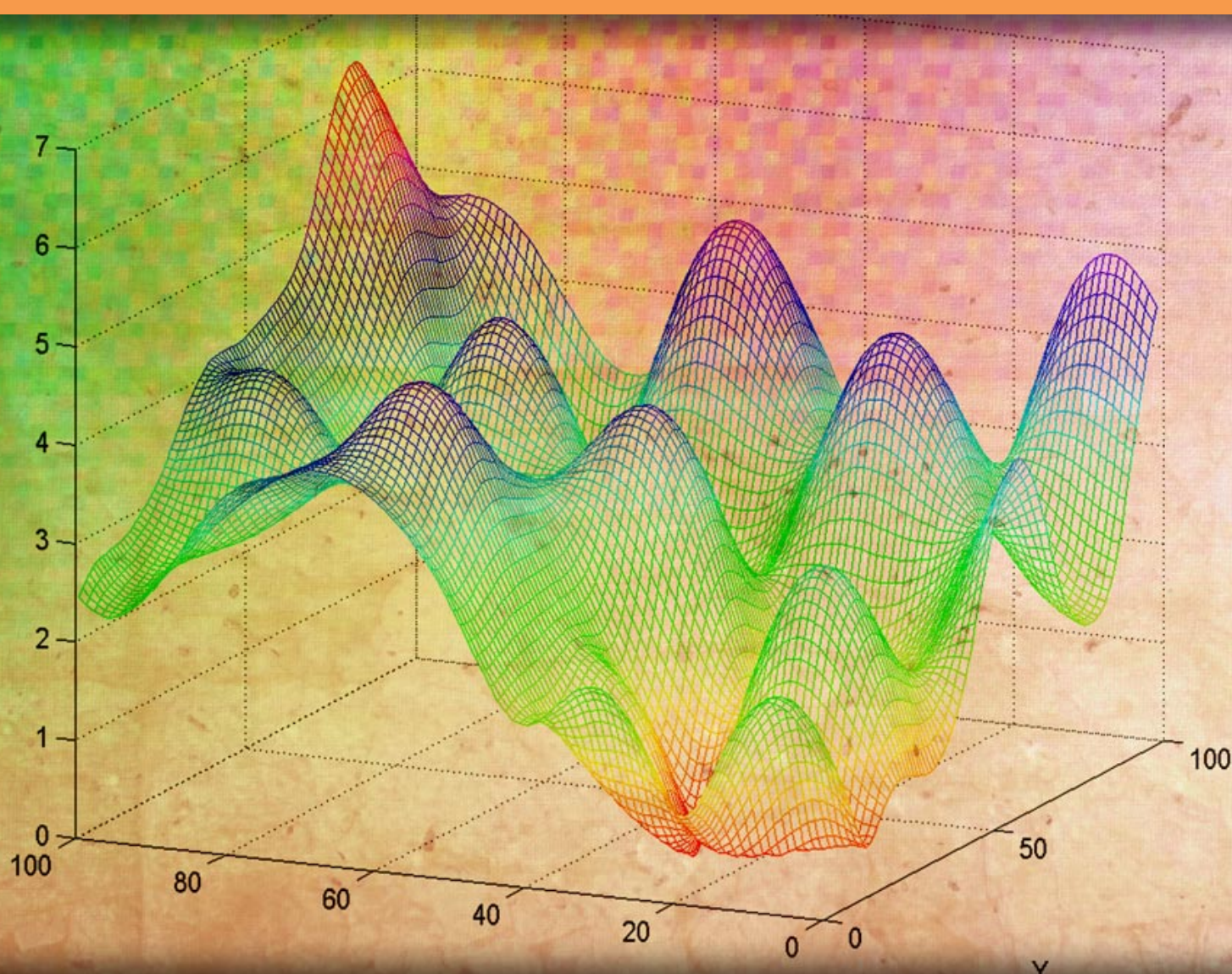


An Introduction to Matlab

Krister Ahlersten



Krister Ahlersten

An Introduction to Matlab



An Introduction to Matlab
© 2012 Krister Ahlersten & bookboon.com
ISBN 978-87-403-0283-7

Contents

	An Introduction to Matlab	9
1	Introduction	10
1.1	Preliminaries and a map of the book	10
	Features of Matlab	12
2	The Desktop	13
3	Some basics of using Matlab	15
3.1	The order of precedence	16
3.2	Some algebraic functions, special characters, and tips	17
3.3	The syntax of functions	19
3.4	Variables	20
3.5	Different types of variables	21
3.6	A note on interpretation and error messages	24
3.7	How Matlab “searches for meaning”	25



Strømmen produseres ofte langt fra der den skal brukes.

Statnett sitt oppdrag er å gjøre strømmen tilgjengelig, uansett hvor i dette langstrakte landet du bor. Det er vi som bygger og drifter "riksveiene" i norsk strømforsyning. Gjennom vårt landsdekkende nett sørger vi for en sikker fordeling av strøm mellom nord, sør, øst og vest.

Vi binder Norge sammen

Statnett
Vårt felles kraftnett

Er du student? Les mer her
www.statnett.no/no/Jobb-og-karriere/Student



4	Matrices, vectors and scalars	27
4.1	Creating matrices	28
4.2	Addressing parts of matrices	31
4.3	Changing parts of a matrix	36
4.4	Some special commands for handling matrices	38
4.5	The Workspace Browser and the Variable Editor	39
4.6	More about matrices	41
5	Mathematical operations with matrices	42
5.1	Functions that operate element-by-element	42
5.2	Elementary mathematical functions that operate columnwise	44
5.3	Matrix algebra	45
5.4	Solving systems of linear equations	50
5.5	Finding linear regression coefficients	51
6	Importing and exporting data	53
6.1	The Current Folder	53
6.2	Problems with importing formatted data	54
6.3	Preparing data to import	54
6.4	Copy-and-paste importing	55
6.5	Importing using the Import Wizard	55

Hva får egentlig en ingeniør- eller teknologistudent for 300 kroner?

- Medlemskap i en aktiv studentorganisasjon – hele studietiden
- 150 tillitsvalgte studenter som ivaretar dine interesser
- Jobbsøkerkurs
- Gratis PC-forsikring og gode bank- og forsikringstilbud
- Teknisk Ukeblad og NITO Refleks
- Møteplasser på web 2.0

Flere medlemsfordeler og innmelding: www.nito.no/student

Alle som studerer på ingeniør-, bioingeniør-, sivilingeniør eller andre teknologistudier (høgskolekandidat, bachelor eller master) kan bli medlem i NITO.

NITO NORGES STØRSTE ORGANISASJON
FOR INGENIØRER OG TEKNOLOGER



6.6	Importing using commands	58
6.7	Exporting to Excel files with commands	61
6.8	More about importing and exporting data	62
7	Graphics	63
7.1	Useful commands for two-dimensional plotting	65
7.2	Time series plotting	66
7.3	Plotting a function	68
7.4	Several graphs in one window and other types of graphs	70
7.5	Other two-dimensional graphs	71
7.6	Plotting tools	72
7.7	More about graphics	74
	Programming in Matlab	75
8	Scripts	76
8.1	The Editor	76
8.2	Writing a script	77
8.3	The search path	79
8.4	User interaction with the script	80



Skatteetaten



Vil du jobbe i et av landets største IT-miljøer?
Vi skal gjøre det kompliserte enkelt

Skatteetaten tilbyr store fagmiljø og utfordrende oppgaver innen:

- > Systemutvikling
- > Service oriented architecture (SOA)
- > Business intelligence (BI)
- > Testledelse
- > Webutvikling
- > IT sikkerhet
- > Infrastruktur
- > Brukergrensesnitt

For nyutdannede IT-spesialister kan vi tilby et to-årig traineeprogram.

For mer informasjon se skatteetaten.no/jobb

Profesjonell • Nytenkende • Imøtekommende



9	User defined functions	82
9.1	About the differences between scripts and user defined functions	84
9.2	More about functions	85
10	Flow control	86
10.1	Loops	86
10.2	Relational and logical operators	88
10.3	Conditional statements	90
10.4	More about flow control	92
11	Numerical analysis and curve fitting	94
11.1	Solving equations	94
11.2	Finding a function minimum point	94
11.3	Numerical integration	95
11.4	Curve fitting	95
11.5	More about numerical analysis	97



OLJE- OG ENERGIDEPARTEMENTET



Er du full av energi?

Olje- og energidepartementets hovedoppgave er å tilrettelegge for en samordnet og helhetlig energipolitikk. Vårt overordnede mål er å sikre høy verdiskapning gjennom effektiv og miljøvennlig forvaltning av energiresursene.

Vi vet at den viktigste kilden til læring etter studiene er arbeidssituasjonen. Hos oss får du:

- Innsikt i olje- og energisektoren og dens økende betydning for norsk økonomi
- Utforme fremtidens energipolitikk
- Se det politiske systemet fra innsiden
- Høy kompetanse på et saksfelt, men også et unikt overblikk over den generelle samfunnsutviklingen
- Raskt ansvar for store og utfordrende oppgaver
- Mulighet til å arbeide med internasjonale spørsmål i en næring der Norge er en betydelig aktør

Vi rekrutterer sivil- og samfunnsøkonomer, jurister og samfunnsvitere fra universiteter og høyskoler.

www.regjeringen.no/oed



Se ledige stillinger her

www.jobb.dep.no/oed



	Debugging and Help	98
12	Debugging	99
12.1	The Code Analyzer	100
12.2	Executing part of the code with F9	102
12.3	Using breakpoints	102
12.4	Checking programs for correct input	105
12.5	Use comments	105
12.6	More on debugging	106
13	Help	107
13.1	To find specific information on functions	107
13.2	To find general information	110
13.3	Online documentation from MathWorks	113
13.4	The internet	114
14	Appendix; Commands used in this book	115
15	Endnotes	117



HELT GRATIS!

S for Skikk & Bank

En bok om ting som er greit å vite når du har flyttet hjemmefra.

dnb.no

DNB

Bank fra A til Å



An Introduction to Matlab

1 Introduction

In many professions, you need programming skills. This book introduces you to a program called Matlab, which is one of the most popular choices for quantitative analysis in fields such as engineering, statistics, economics/finance, and artificial intelligence, to name a few. It will give you the tools to get started whether you are a student, a researcher, or a practitioner.

This is a beginner's book and it does not assume that you have any previous knowledge of programming. Basic concepts are described at length and, particularly in the beginning, we explain step-by-step how to start using the program. However, we do assume that you have Matlab installed and that you have at least an average level of general computer experience. Having used Excel, or some other spreadsheet program, is of some help in learning Matlab. Knowledge of statistical software, such as Stata, SAS, or EViews, is also beneficial, although Matlab is not a statistics program. One difference is that these programs largely rely on predefined routines for statistical analysis, whereas Matlab often requires you to define your own routines. Another major difference is that matrix algebra is at the heart of Matlab, but peripheral and seldom used in the other programs. Previous knowledge of matrix algebra is, consequently, very useful when learning Matlab. Naturally, we also assume that you have mathematical knowledge corresponding to university level.

In this book, you will learn how to solve simple quantitative problems using Matlab. This includes how to load data into the program, how to do your own programming, and how to present results in graphs. Last, you will learn some error checking techniques and how to find help.

In a way, learning programming is just like learning any other language. It can be used for many purposes, including just having fun. When you know a language well enough, it is joyous and almost effortless to use it. In addition, you can use it as a tool to solve interesting and complex problems. However, when you are learning programming, you use computers to solve daft and boring problems in a time consuming way. And this is frustrating. Do not let the frustration that you will undoubtedly feel stand in way of your learning.

1.1 Preliminaries and a map of the book

Matlab is short for Matrix Laboratory and was originally a tool for performing matrix algebra. Over time, it has evolved into a programming environment with several parts. In this book, we will focus on some desktop tools, basic mathematical functions, two-dimensional graphics, and how to write programs. However, we will not present features that are more advanced. Among these are three-dimensional graphics, graphical user interfaces, interaction with other software, and toolboxes. Toolboxes are packages of programs that provide additional functionality to Matlab. Examples are Optimization toolbox, for function minimization, and Statistics toolbox, for statistical functions. No toolboxes are needed for this book.

It should be noted that Matlab is a living program that evolves over time. Since the early 2000s there have been updates twice a year. This means that some details may have changed since this book was written, although there are seldom changes to the basics that we describe here. (We use issue R2011a with the Windows 7 operating system.)

Furthermore, there are often many different ways to do things in Matlab. There are, for instance, at least three different ways to find the coefficients of a simple linear regression, and many tasks that can be performed using drop-down menus can also be performed using keyboard shortcuts or using commands. In most cases, we only describe one way to perform tasks. At the beginning of the book, there are, however, several descriptions of how you can use alternative ways to do things, but as we go along, these become fewer and fewer. Often, the alternative ways are similar between tasks, and once you have seen a few examples, you will be able to figure out the alternatives on your own.

As we begin, it is good to know where we are heading. We will move through the following topics:

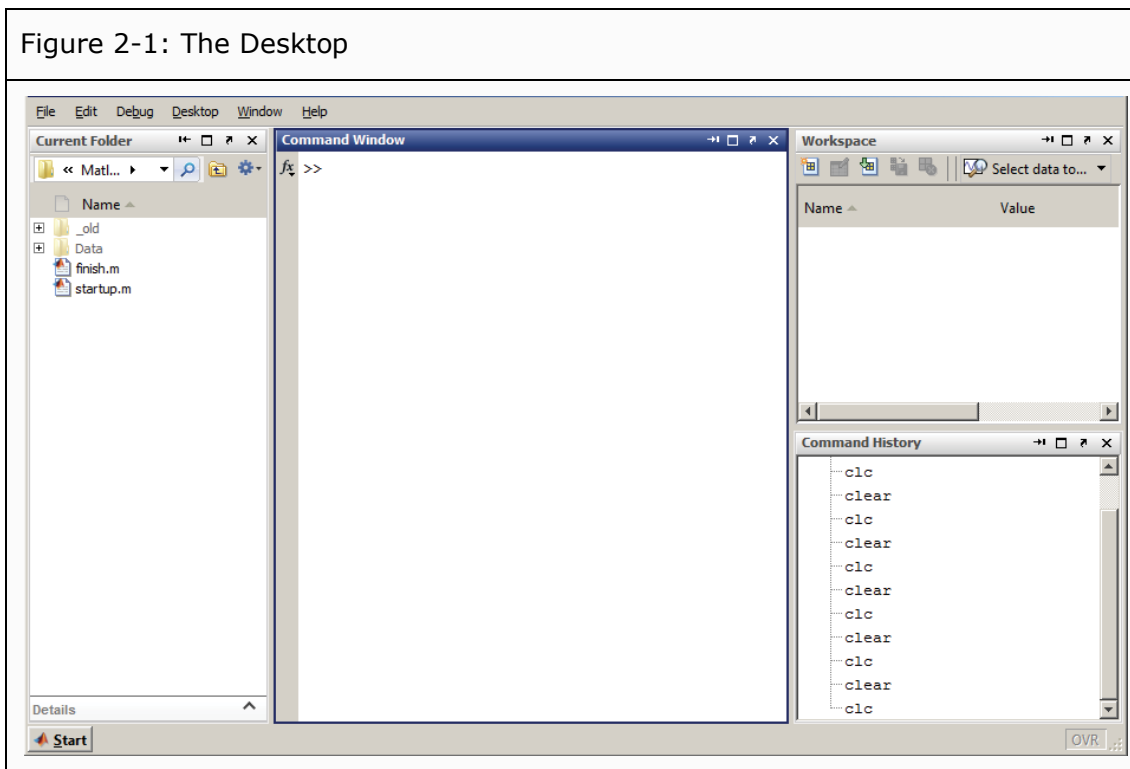
- It is necessary to understand the different parts of the **Desktop** (i.e., what you see when you open the program), and what they can be used for.
- Matlab can be used for solving **algebraic problems**. This is a good way to become familiar with the program and the basic syntax.
- Information is stored as **variables** and these can be of different types. We describe the most common ones.
- **Matrix manipulation** techniques are important in Matlab and we show many examples. After having discussed **matrix algebra**, we show how to use it to solve systems of linear equations and find linear regression coefficients.
- Data must often be **imported** into the program and **exported** out of it, before and after analysis. We look at different ways of doing this.
- Data and results are often presented visually, so we look at some ways to produce two-dimensional **graphs**.
- By this time, we are ready to start writing routines; programs. Matlab distinguishes between **scripts** and **functions** and we look at the differences between these and at some programming techniques.
- After we have learnt how to write functions, we can perform some **numerical analysis**, such as solving equations, finding a function minimum, and integrate a function.
- After programming comes error checking, **debugging**, and searching for **help**.

Features of Matlab

2 The Desktop

The first time you open Matlab, you will probably see something like what you see in Figure 2.1. If this is not what you see, you can change it into something similar by choosing `Desktop > Desktop Layout > Default` in the drop-down menus.¹ Much of what you see on the Desktop is nice-to-have but far from necessary, so you do not need to learn everything you see; although you might eventually want to do so. Let us go through some of the features on the Desktop that you are most likely to use.

Figure 2-1: The Desktop



The Desktop is, in the default case, divided into four different windows. Each of these has a name, indicating its function, at the top-left and a few symbols at the top-right. (If you do not see the names, choose `Desktop > Titles`.) The four windows are:

- *Command Window*

This is the most central part of Matlab. From here, you enter commands that tell the program what you want it to do.

- *Current Folder*

This window is similar to Explorer in Windows (i.e., the program you use when moving up and down through folders). In this case, there are four objects in the Current Folder, `_old`, `Data`, `finish.m` and `startup.m`. Just below the name of the window, there are also some tools to navigate to other folders.


- *Workspace*




The Workspace lists all the existing variables. Since we have not yet defined any variables, the window is empty. You can also view and change variables here.

- *Command History*

This is a list of commands previously entered into Matlab. In Figure 2-1, the user has previously entered the commands `clear` and `clc` several times.

There are more windows/tools than the four described here. If you click the `Desktop` drop-down menu, you see a list of different tools you can use. The ones that are in use are checked. Tools that we will discuss later include `Help`, `Editor`, `Figures`, and `Variable Editor` (see sections 13, 8.1, 7, and 4.5).

The Desktop is very dynamic and you can change the layout in many ways to suit your personal preferences. Let us look at the four symbols in the upper-right corner of each window: .

- If you click the first of the symbols, the window will disappear and instead a corresponding label will appear at the left or at the right side of the Desktop, depending on where the original window was located. If you click the label, the window will be temporarily restored, but it will disappear again as soon as you move away from it. This is convenient if you only use the tool seldom, but still want quick access to it. To restore the window completely, click the symbol  in the upper-right corner of the window after having clicked the label.
- Click the second symbol, and all other windows disappear so that you only see the one whose symbol you clicked. To restore, click . This feature is convenient if you temporarily need to get a better overview of a particular window.
- Clicking the third symbol causes the window to undock from the rest. You can then move the window independently of the other ones. Click  to dock again.
- The forth symbol is for closing the window permanently. To restore it, use the `Desktop` drop-down menu and choose the tool from the list.

In addition to choosing which tools/windows that appear on the Desktop, you can choose the layout. To change the size of a certain window, click-and-drag the grey borders between the windows. Furthermore, you can move the whole window by clicking the list where the name label appears and then dragging it to another location. For example, if you want the Command Window to be located beneath the Current Folder instead, you click the Command Window name list and drag it to somewhere in the lower parts of the Current Folder. As you drag the window, Matlab highlights which area it would occupy if you drop it at a certain point.

It is a good idea to play around with these features for a while to get a sense of what is possible. If you arrive at a layout you like, you can save it by choosing `Desktop > Save Layout...` and then choosing a name for your layout. Then you can later revert to this layout by choosing `Desktop > Desktop Layout` and, finally, choosing the name you have given the layout.

3 Some basics of using Matlab

To begin, you can use Matlab for simple arithmetic problems. Symbols like + (plus), - (minus), * (multiply), and / (divide) all work as you would expect. In addition, ^ is used for exponentiation.

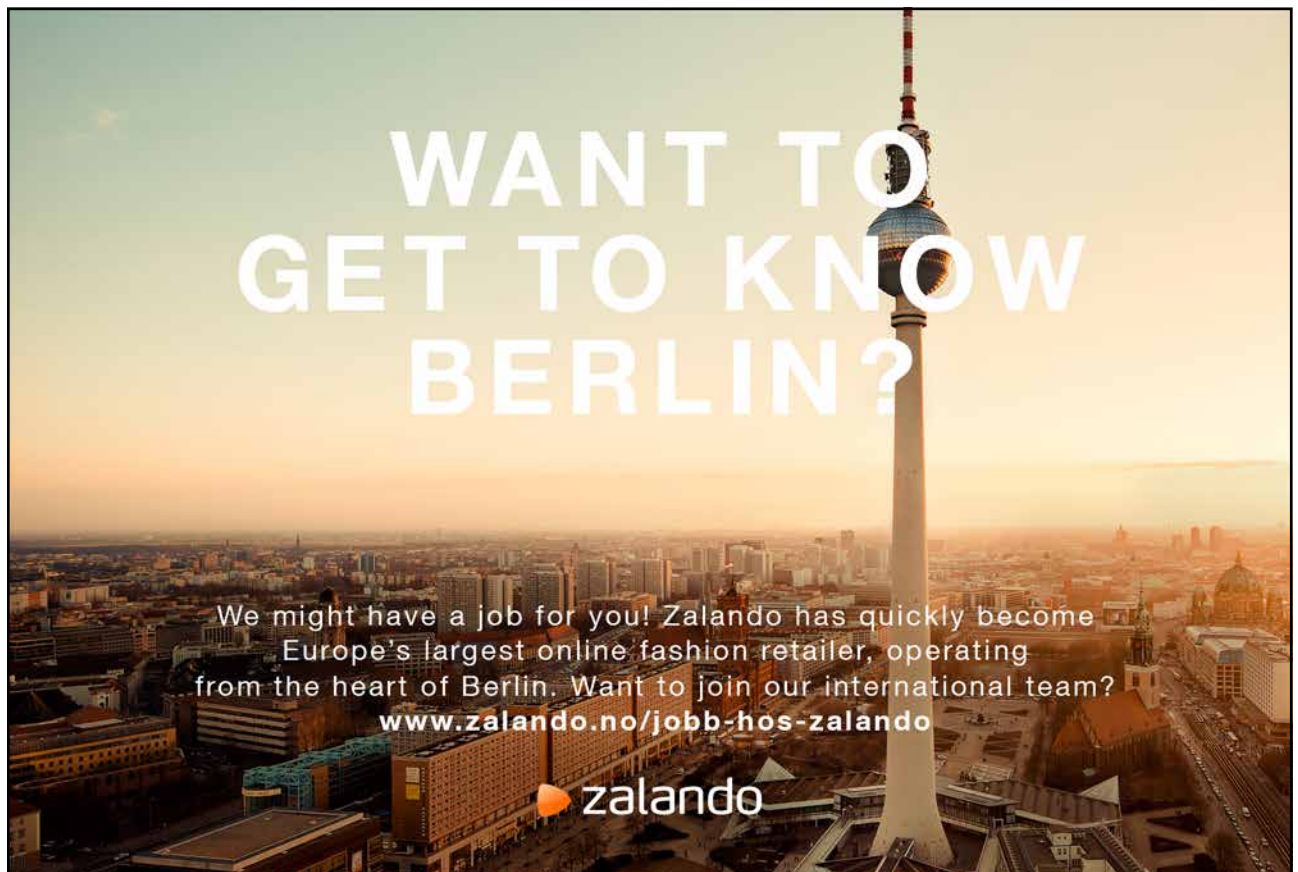
For example, if you type

```
>> 75 - 32 * 2 + 4 / 2
```

in the Command Window (note that the >> is added by Matlab²) and press <Enter>, Matlab calculates the value of the expression and responds with

```
ans =  
    13
```

There are a few things here to note. First, when you enter a command to calculate a value, Matlab performs the calculations and puts the result in a variable named `ans` (for answer). If you check the Workspace, there is now a variable listed there: `ans`. Second, by default Matlab prints the value of the variable in the Command Window in two rows. In the first row, we see the variable name, `ans` in this case, and an equality sign, and in the second, we see the value of the variable, which in this case is 13. The value is indented by a few blank spaces.



3.1 The order of precedence

It is important to note the order in which algebraic expressions are calculated. This is called the order of precedence, or the order of operations. As you probably expect, multiplication and division are calculated before any addition or subtraction. In the expression above, this means that $32 * 2 = 64$ is calculated first, $4 / 2 = 2$ is calculated second, and then $75 - 64 + 2 = 13$ is calculated last.

In general, the order of precedence is

1. parentheses
2. exponentiation
3. multiplication, division
4. addition, subtraction

If the list is not enough to determine the order in which to calculate an expression, the calculations are done from left to right. For example, if you enter $10/20/10$, Matlab first calculates $10/20 = 0.5$ and then $0.5/10 = 0.05$. It does *not* start by calculating $20/10 = 2$ and then $10/2 = 5$. If you want the latter, you have to make this clear by adding parentheses: $10/(20/10) = 5$. If you try it in Matlab, you get

```
>> 10/20/10
ans =
    0.05
```

and

```
>> 10/(20/10)
ans =
    5
```

If you are unsure, always add parentheses. That also makes many expressions easier to read. Writing

```
>> 75 - (32 * 2) + (4 / 2)
ans =
    13
```

does not change the outcome, but you could argue that it is somewhat easier to read. As expressions become more complex, the difference becomes more pronounced.

3.2 Some algebraic functions, special characters, and tips

In Matlab, you usually have to enter all instructions for what you want the program to do as commands. This means that it is necessary to know some basic commands and syntax.

A few characters have special meaning and are used frequently.

- , Comma
This can be used for separating commands that are written on the same line. You can, for example, type `sqrt(4), sqrt(9)` to have Matlab calculate both values at once.
- ; Semicolon
This suppresses output when commands are executed.
For example, if you type `9+5;` (including the semicolon) and press <Enter>, the expression is evaluated and saved in the `ans` variable (which you can check in the Workspace). However, nothing is displayed in the Command Window. This is useful when you run programs and do not want intermediary calculations to be displayed.
Note: If you separate commands with `;` you do not have to use `,` as well.
- % Comments
Everything after the %-character until the end of the line is ignored. This is useful for adding comments in programs but is seldom used in the Command Window.

Basic algebraic functions include

<code>sqrt(9)</code>	Square root of 9.
<code>3^4</code>	Exponentiation 3^4 ; 3 to the power of 4 (i.e., $3*3*3*3$).
<code>exp(4)</code>	e^4 ; exponentiation with base e (≈ 2.7183).
<code>log(5)</code>	The natural logarithm of 5 (note: with base e , <i>not</i> with base 10).
<code>log10(5)</code>	The logarithm of 5, using base 10.
<code>abs(-3)</code>	The absolute value of -3 .
<code>round(1.3)</code>	Round to the nearest integer.
<code>floor(1.3)</code>	Round to the nearest smaller integer (i.e., towards minus infinity).
<code>ceil(1.3)</code>	Round to the nearest larger integer (i.e., towards plus infinity).
<code>fix(1.3)</code>	Round to the nearest integer towards zero (i.e., downwards for positive numbers and upwards for negative).
<code>sign(1)</code>	The sign of 1: -1 if < 0 ; 1 if > 0 ; and 0 if $= 0$.

Note that you do not have to use functions one at a time. Instead, you can nest them. If you, for example, want to calculate the absolute value of -4 to the power of 3 , and then round that off to the nearest integer, you can issue

```
>> round(abs(-4)^3)
ans =
    64
```

The calculations start in the innermost pair of parentheses and move outwards.

In addition, you will probably want to use the following

help	We will discuss how to find help later (see Section 13). Here, it is enough to note that you can get help on any function by typing <code>help</code> and the function name. For example, <code>help sign</code> will produce a short text on how the <code>sign</code> function works in Matlab. In addition, you get clickable links to related functions and a clickable link to the corresponding reference page in the documentation.
clc	Clears the Command Window. Only the window is cleared, no variables or anything else is changed. This is good for getting rid of clutter.
clear	Deletes variables. <code>clear</code> alone deletes all variables, <code>clear</code> and one or several variable names, for example <code>clear abc</code> , deletes only the named variable(s).
↑	Pressing this arrow-key (typically located on the lower right side of the keyboard) brings back the previous command lines one-by-one. If you type one or several characters first and then press the arrow key, Matlab skips to those lines that begin with the same characters. For example, if you type <code>75</code> and press ↑, you get back the line <code>75 = (32*2) + (4/2)</code> if you entered it earlier.
format	Changes the way that output is displayed in the Command Window. Note: it does not change the internal calculations or the value of the variable, only how output is displayed in the Command Window. <code>format loose</code> adds extra lines between output and <code>format compact</code> suppresses extra lines. <code>format long g</code> displays numbers using 15 digits and <code>format short g</code> displays them using 5 digits. For example, $1/3$ is displayed as <code>0.333333333333333</code> in the first case and as <code>0.33333</code> in the second. Try <code>help format</code> to get a few more tips on how to use this command.

3.3 The syntax of functions

Most functions in Matlab have a common form. They take one or several arguments as input and they produce one or several output arguments. The general syntax of a function is

```
[output1, output2, ...] = function_name(input1, input2, ...)
```

Input arguments are enclosed within parentheses, and if there are more than one argument, they must be separated by commas. If there is only one output argument, no square brackets are needed, but if there are more than one they must be added. Output arguments can be separated by commas or by blank space. Note that function names are almost always in lower case.

Some functions, such as plus and minus, have alternative syntaxes. For plus, for example, you can issue either `2+3` or `plus(2,3)` (see also sections 5.1, 5.3.2, and 5.3.3).

Furthermore, certain functions work differently depending on how many input arguments you enter or how many output arguments you request. See Section 4.1.1 for an example using the function `diag()` or Section 4.4 for an example using the function `find()`.



"I studied English for 16 years but...
...I finally learned to speak it in just six lessons"

Jane, Chinese architect

ENGLISH OUT THERE

Click to hear me talking before and after my unique course download



3.4 Variables

Earlier we calculated

```
>> 10/20/10
ans =
    0.05
```

Let us go through what happens when this command is issued, step-by-step. We enter the expression `10/20/10`. Matlab calculates that the value of this is `0.05`. Then it defines a variable with the name `ans` and sets the value of this variable equal to the answer (i.e., equal to `0.05` in this case). If a variable named `ans` already exists, the value is changed to the answer. Then it prints the variable name, `ans`, in the Command Window followed by an equality sign and, finally, the value, `0.05`, in the next row. The latter part, the printing in the Command Window, would have been omitted if we had ended the expression, `10/20/10`, with a semicolon.

This process is a special case of defining a variable that occurs when the user has not specified a variable name. If, instead, you enter `abc = 10/20/10` the process is the same, but instead of defining a variable named `ans` Matlab creates a variable named `abc`, assigns the value of the answer to this variable and prints the information in the Command Window, so that what is seen there is

```
>> abc = 10/20/10
abc =
    0.05
```

Since the equality sign, in this context, tells Matlab to *assign* a value to a variable, it is called *the assignment operator*. The command is interpreted as “set the variable...equal to...” After you have issued the command above, `abc` has been assigned a value of `0.05`. Checking the Workspace, you can confirm that there is now a variable named `abc` listed there.

When a variable has been defined, you can ask for its value by entering the name:

```
>> abc
abc =
    0.05
```

Variables can also be reassigned. The variable `abc` has the value `0.05`, but if you type `abc = 3` the old value is replaced by the new one. You can also reassign variables using other variables or even using the variable itself. So, to double the value of `abc` (which is now `3`), you can enter

```
>> abc = abc*2
abc =
    6
```

Variable names must begin with a letter and can be at most 63 characters long.³ It is a very good idea to use names that describe what the variables represent. Multi-word names are more readable if you use capital letters or underscores to indicate new words, for example `interestRate` or `exchange_rate`. Note that Matlab is case sensitive, so `c` and `C` are two different variables, as are `interestRate` and `interestrate`.

3.4.1 Predefined variables

A few variables are predefined in Matlab.

<code>ans</code>	“answer” A variable that holds the (latest) value that has been calculated with no specified variable name. If this variable is deleted (for example by entering <code>clear ans</code>) it reappears as soon as you perform a new calculation with no specified variable name.
<code>pi</code>	The constant $\pi = 3.141592653589793\dots$
<code>inf</code>	Infinity Rather than infinity this means “overflow”. This can be the output of a calculation. For example, <code>1/0</code> gives the result <code>inf</code> . Note that you can also use it in calculations. For example, <code>3*inf</code> is <code>inf</code> .
<code>NaN</code>	Not-a-number This is used when it is not possible to record a valid numerical value, for example when you have missing observations in a data set. It can also be used in calculations or be the outcome of a calculation. <code>NaN*2</code> and <code>0/0</code> both resolve to <code>NaN</code> .

3.5 Different types of variables

Variables can be of different types. We need to distinguish between numerical, character, and logical variables. Later, we will also briefly present cell variables.

3.5.1 Numerical and character variables


It is important to note that there are different *types* of variables. `abc` above is a *numerical* variable. However, variables can also hold one or several characters. To tell Matlab that you want it to interpret something as a string of characters, you enclose it within single quotes (`'`). Enter, for example, the text `'def'` within single quotes in the Command Window:

```
>> 'def'  
ans =  
def
```

Part of the text is color coded violet. This is to highlight that this part is a character string.⁴ Matlab interprets your command as though you want to evaluate a string of characters. And that produces the same string of characters, `def`, which is then assigned to the variable `ans` and printed in the Command Window. If you check the Workspace, you can see that the `ans` variable is now labeled "char" in the class column to indicate that it is a character variable. Numerical variables are usually labeled "double" (referring to how numbers are stored internally).

Consequently, it is very different to enter `123` and to enter `'123'`. If we enter both, separating them with a comma, we get

```
>> 123, '123'  
ans =  
    123  
ans =  
    123
```



WHILE YOU WERE SLEEPING...

www.fuqua.duke.edu/whileyouweresleeping

DUKE
THE FUQUA
SCHOOL
OF BUSINESS



The two answers look deceptively similar. The main difference is that the first `123` has a few white spaces in front of it, so that it is located a bit off from the left side. The answers are, however, completely different. In the first case, `123` is interpreted as a number and evaluated as such. The result is, naturally, the number `123`. When numerical values are printed to the Command Window, a few spaces are inserted to the left. You can see this in all the preceding examples where numbers were printed.

In the second case, `'123'`, is interpreted as a string of characters. The fact that a string of 1, 2 and 3 can be interpreted as a number is ignored. The string of characters is evaluated and the result is just the same string of characters. The result is assigned to the variable `ans` and printed to the Command Window. However, as it is not a number, no white spaces are inserted in front of it. The string is therefore printed immediately next to the window edge.

There are convenient ways to translate numerical variables to character strings and vice versa.

<code>str2num</code>	Converts a string of characters to a number, given that Matlab can interpret the string as a number. For example, <code>str2num('542')</code> is 542.
<code>num2str</code>	Converts a number to a string of characters. For example, <code>num2str(542)</code> is the string <code>'542'</code> .

Sometimes, for instance when adding a title to a graph or when printing information to the Command Window, it is useful to mix numbers and text. In such cases, numbers must first be “translated” into character strings. This is described in Section 8.4.

Strings have several different functions in Matlab. They are frequently used to display messages and in addresses to files on the computer, for instance when importing data. Some functions also take string input arguments. You can also enter formatting commands for plots as strings.

3.5.2 Logical variables

Logical variables are a type of variable that can only take the values *true* and *false*. In Matlab, *true* is represented as 1 and *false* is represented as 0. Superficially, logical variables therefore look like ordinary numerical variables. Logical variables are usually the outcome of relational or logical statements. Suppose we take the variable `abc` (that equals 6) from earlier and issue the statement

```
>> abc > 5
ans =
    1
```


The answer is 1, meaning true, since it is true that `abc` is greater than 5. Note now that, in this case `ans` is not an ordinary numerical variable. Check the Workspace to see that the class of `ans` is "logical". Does this matter? The answer is that, sometimes it does, sometimes it does not. The variable `ans` can still be used in algebraic expressions. Then it is treated as an ordinary zero or one, depending on its value. If we issue

```
>> ans*1
ans =
    1
```

the variable looks unchanged (i.e., its value is still 1). If, however, we check the Workspace, we see that the class has changed to "double", indicating that it is now an ordinary numerical variable. In algebraic expressions, it does not matter whether the variable is logical or numerical. However, we will later see that it does matter when the variable is used for addressing purposes (see Section 4.2.3).

3.6 A note on interpretation and error messages

It is important to understand that Matlab *interprets* everything that you enter into the Command Window. To get the response that you want, you have to enter commands that Matlab interprets in the way that you mean them. It is not unusual that the user enters commands that seem obvious to him or her, but gets a response that seems weird or gets an error message. If you, for example, enter `def` (without single quotes), and have not defined this as a variable, Matlab is unable to interpret your command and responds by issuing an error message:

```
??? Undefined function or variable 'def'.
```

As you can see from the message, Matlab tried to interpret what you entered either as a function or as a variable, but was unable to find any interpretation for it as either. Error messages are, by default, color-coded red.

The error messages are often informative and you can use them to understand what it is that Matlab does not understand. However, this is not always so. If you, for example, entered `def` (forgetting the single quotes) and wanted Matlab to interpret this as a string of characters, the error message above is a bit perplexing. It just tells you that Matlab was not able to understand your command either as a function or as a variable. In such a case, you can often understand the message as though you performed a more fundamental error: You have entered a command that to Matlab looks like a function or like a variable. If that was not what you wanted, you have to look for a different way to enter your command. In this case, by enclosing the `def` within single quotes.

3.7 How Matlab “searches for meaning”

One curious feature of Matlab is that it is possible to use a function name as a variable name. For example, `sqrt()` is a function for calculating the square root:

```
>> sqrt(4)
ans =
     2
```

Suppose that you want to assign this answer, 2, to a variable and that you think that `sqrt` is a good name.

```
>> sqrt = sqrt(4)
sqrt =
     2
```

This is perfectly ok, although not a good practice. What happens if you do this is that the variable takes precedence over the function, which, consequently, cannot be used any more. Entering the following commands sheds some light on this:



Vi vokser i Norge
og har virksomhet
helt frem til 2050

Er du interessert i sommerjobb
eller fast stilling?

Se informasjon om sommerjobber på
www.bp.no



```
>> sqrt, sqrt(1), sqrt(4)

sqrt =
     2
ans =
     2

??? Index exceeds matrix dimensions.
```

In the first case, `sqrt` produces `sqrt=2` which is the value of the variable `sqrt`. In the second case, `sqrt(1)` produces `ans=2`, which is not the answer we want, if we want to calculate the square root of 1. In the third case, `sqrt(4)` produces a counterintuitive error message.

The strange results are all consequences of the fact that `sqrt` is now a variable, not a function. In the first case, Matlab interprets the command `sqrt` as if you are asking for the value of the variable, which is 2. In the second case, it interprets it as if you are asking for the first element of the variable `sqrt`, which is also 2. In the next section, where we talk about matrices, we will see that a variable can hold more than one element. In the third case, Matlab interprets the command as if you are asking for the value of the fourth element of the variable `sqrt`. However, `sqrt` has only one element, so you get an error message.

To be able to use the function again, you must first delete the variable. To do that from the Command Window, use the command `clear` (see Section 3.2).

```
>> clear sqrt
```

It is also possible to clear specific variables by selecting them in the Workspace and pressing <Delete> or by right-clicking the variable and choosing `delete` in the context menu.

It is, of course, best to avoid using function names as variable names. If you are unsure if a certain name is already in use, just issue the command `exist` and the name. If it is not in use, Matlab will respond with 0. Then you know that it is ok to use that name for a variable.

So, why does the variable take precedence over the function? This has to do with the order in which Matlab “searches for meaning”. When a command is issued, Matlab first searches the Workspace to see if there is a variable by that name there. If there is, the program returns the value of the variable. If no variable is found, Matlab searches the Current Folder (see Section 2 and Section 6.1). If a function file is found there, it is selected and executed. If, on the other hand, nothing is found in the Current Folder either, Matlab starts searching in folders along a predefined search path (see Section 8.3) to see if a function can be found there. Again, if one is found, it is selected and executed. If nothing is found along the path either, Matlab gives up and issues an error message. This is why the first error message we encountered was that the program could not find a defined variable or function.

4 Matrices, vectors and scalars

One defining feature of Matlab is that numerical variables are typically matrices, not scalars. A scalar is a single number such as 5, 7.89, or 10243. All the numerical examples so far have used scalars. A matrix, on the other hand, is a rectangular set of numbers, arranged in rows and columns. In text, a matrix is usually enclosed within brackets. As an example, consider the following matrix that has three rows and four columns:

$$\begin{bmatrix} 1 & 32 & 7 & 8 \\ 2 & 4 & 1 & 9 \\ 3 & 19 & 3 & 9 \end{bmatrix}$$

The size of the matrix is measured by the number of rows and columns. This matrix is a 3x4 matrix (pronounced three-by-four). As a special case, you may consider a scalar to be a 1x1 matrix. Matrices with only one column or one row often have special meaning, and are usually referred to as column vectors and row vectors. Furthermore, a matrix with equally many rows and columns is often called a square matrix.

Oftentimes, the columns and rows of a matrix represent some distinguishing feature of the data. In the example matrix, the first column might denote the day the observations were made, numbered from one to three. The second to fourth columns might be certain outcomes that occurred during the corresponding days. For example, it could be the amount of rain in three different locations measured in, say, millimeters. The amount of rain in the second location on the third day would then be 3 millimeters. The important feature is that the order of the numbers represents relations between them. If you are familiar with spreadsheet programs, such as Excel, you can think of a matrix as a rectangular set of cells in a spreadsheet. Note, however, that it is not possible to have “empty” entries in a matrix. If no value is appropriate for a certain entry, you use NaN (not-a-number).

Similar to scalars, matrices can be added, subtracted, multiplied, etc. This is called matrix algebra. Matlab does matrix algebra really well. While you do not have to know any matrix algebra to work with Matlab, some of its power is lost if you do not. And you still have to know how to create and change matrices, as they are used for several different purposes in Matlab. For instance, in Section 6, you will see that when data is imported into Matlab, it is typically imported as a matrix.

In this section, we will describe how to create and change matrices, as well as how to address parts of them. Section 5 describes mathematical operations with matrices, including some matrix algebra.

4.1 Creating matrices

The most straightforward way to create a matrix in Matlab is to enter it element-by-element. Say we want to define a variable, `testMatrix`, which is equal to the matrix in the example above. We then enter the variable name followed by the assignment operator and a left square bracket, often produced by typing `<Ctrl>-<Alt>-` (`[`). We then enter the first row of numbers, separated by one or several blank spaces or, alternatively, separated by commas. To indicate a new row we enter a semicolon, and then continue with the numbers on the second row, etc. Lastly, we enter a closing right square bracket. What we see in the Command Window is

```
>> testMatrix = [1 32 7 8 ; 2 4 1 9 ; 3 19 3 9]
testMatrix =
     1    32     7     8
     2     4     1     9
     3    19     3     9
```

When Matlab prints the matrix in the Command Window, it does not print any brackets. Now, `testMatrix` is defined as a variable and it is listed in the Workspace. Its class is "double", similarly to other numerical variables. However, in the value-column you do not see its value. Instead it states `<3x4 double>`, where the 3x4 refers to the size of the matrix.



Entering matrices manually is only practical when they are sufficiently small. Surprisingly often, however, useful matrices have systematic features that make them easy to create in other ways. They might contain only zeroes or only ones, or they might contain equally spaced ascending numbers. For these purposes, it is very useful to know the commands listed below.

4.1.1 Commands for creating matrices

<code>[... ; ...]</code>	Typing all values manually within square brackets and indicating a new row with a semicolon. For example, <code>[1 2 3 ; 4 5 6]</code> .
<code>1:3:20</code>	<p>The colon operator</p> <p>This can only be used for creating a matrix with one row; a row vector.</p> <p>The first element will be equal to the first number (1 in the example). Subsequent elements will increase by the number in the middle (the increment; 3 in the example) and the last element will be no higher than 20.</p> <p>The resulting matrix in the example is <code>[1 4 7 10 13 16 19]</code>. 20 is not included as the next element would have been $19+3=22$, which is greater than 20.</p> <p>If the middle number is omitted, Matlab uses an increment of 1. Consequently, <code>4:8</code> is the matrix <code>[4 5 6 7 8]</code>.</p>
<code>linspace(2,9,5)</code>	<p>Linear spacing</p> <p>This creates a matrix with only one row; a row vector. The first element is 2, the last element is 9, and there are 5 linearly spaced elements (i.e., the distance between each is the same).</p> <p>In the example, the resulting matrix is <code>[2 3.75 5.5 7.25 9]</code>. Note that the difference between each consecutive number is 1.75 and that there are 5 elements.</p>
<code>zeros(2,3)</code>	Creates a 2x3 matrix of zeros.
<code>ones(2,3)</code>	Creates a 2x3 matrix of ones.
<code>eye(4)</code>	Creates a 4x4 square matrix with a diagonal of ones and all other elements equal to zero.
<code>rand(5,4)</code>	Creates a 5x4 matrix of random numbers between 0 and 1 (drawn from a uniform distribution).
<code>randn(5,4)</code>	Creates a 5x4 matrix of random numbers between minus infinity and plus infinity (drawn from a standard normal distribution, meaning the numbers will usually be between -3 and 3).
<code>repmat(A,2,3)</code>	Repeats the matrix A twice vertically and three times horizontally.

`[A B], [A;B]`

Concatenation

Assuming that A and B are already defined matrices, this produces a matrix with elements equal to those of A and B. In the first case, the matrices are lined up beside each other, and in the second, they are stacked on top of each other. Note that, A and B must have the same number of rows in the first case, and the same number of columns in the second.

`diag(X)`

Diagonal

This command has different meanings depending on the input matrix, X.

If X is a matrix and not a vector (i.e., if it has at least two rows and two columns), the command creates a row vector containing the values of the diagonal of X, beginning with the top-left element.

if X is a vector, the command creates a square matrix with the values of X on the diagonal and zeros elsewhere.

To see an example of some of these commands, consider the following. (Note that the semicolons suppress all output except the last one.)

```
>> A = 2:2:6; B = linspace(10,16,3); C = zeros(1,3);
>> D = [A ; B ; C]; E = [D eye(3)]
E =
     2     4     6     1     0     0
    10    13    16     0     1     0
     0     0     0     0     0     1
```

A is defined using the colon operator, and is a row vector with the first element equal to 2, the last element no higher than 6, and with an increment of 2 for each element: `[2 4 6]`. B is defined as a row vector with 3 linearly spaced elements, starting with 10 and ending with 16: `[10 13 16]`. C is a 1x3 matrix of only zeros: `[0 0 0]`.

D is defined by concatenating the first three variables on top of each other (since they are separated by semicolons). Finally, E is defined by concatenating D and a 3x3 matrix with ones on the diagonal and zeros elsewhere. The concatenation is horizontal since there are no semicolons between the input matrices. The resulting matrix, E, is displayed in the Command Window.

Lastly, you may note that semicolons have a different meaning in matrices (i.e., when between square brackets) from the one we first learnt. At the end of commands, they suppress output in the Command Window; within a matrix, they indicate new rows.

4.2 Addressing parts of matrices


Sometimes we want to pick out, or change, a subset of the elements of a matrix. There are three different ways of addressing a subset. The first is to use the row and column numbers, the second is to use one single index, and the third is to use conditional statements.

4.2.1 Addressing using row and column numbers

Say we have the matrix `testMatrix` defined earlier. We noted that, for the second location in the third day it rained 3 millimeters (i.e., the corresponding element is 3). To pick this number out from the matrix, note that it is located on the third row and in the third column. Addressing this particular location within the matrix is done by entering the row and column numbers within parentheses after the variable name:

```
>> testMatrix(3,3)
ans =
     3
```

You can also pick out a larger subset of a matrix. Suppose, for instance, that we only want the day numbers and the observations from the third location (i.e., columns one and four). This can be addressed as



The advertisement features a background image of a person running on a path during a sunrise or sunset. The GaiTEYE logo is in the top left, with the tagline 'Challenge the way we run'. The main text reads 'EXPERIENCE THE POWER OF FULL ENGAGEMENT...'. Below this, separated by a dotted line, are the phrases 'RUN FASTER.', 'RUN LONGER..', and 'RUN EASIER...'. In the bottom right, there is a yellow button with the text 'READ MORE & PRE-ORDER TODAY' and 'WWW.GAITEYE.COM', with a hand cursor icon pointing at it.



```
>> testMatrix([1 2 3],[1 4])
ans =

     1     8
     2     9
     3     9
```

Here, the addressing of the rows and columns of the matrix is done using vectors. The first vector picks out rows 1, 2, and 3, and the second picks out columns 1 and 4. Oftentimes, it is convenient to use the colon operator to create the addressing vectors. In the example, we wanted each row from row 1 to row 3. This can be written as `1:3`, since this creates a row vector starting with 1, ending with 3, and increasing with 1 in each step. (As we have not entered any increment, Matlab assumes an increment of 1.)

If you want to pick out *all* the elements in one dimension (i.e., all rows or all columns), there is an even simpler way to do that. Note that, in the example we pick out all rows from `testMatrix`. In such cases, you may enter a colon, without any start or end indicators. Both of these two methods produce the same result as the previous example.

```
>> testMatrix(1:3,[1 4]), testMatrix(:,[1 4])
ans =

     1     8
     2     9
     3     9

ans =

     1     8
     2     9
     3     9
```

Oftentimes, we want to pick out all elements from some starting row, or column, until the last one. Then there is another simplifying trick where you can refer to the last row or column as `end`. For example, if we want all rows except the last one, this can be addressed as

```
>> testMatrix(1:end-1,[1 4])
ans =

     1     8
     2     9
```

This is, for instance, useful if we want to lag a matrix one or several periods and we do not know the number of rows or columns, as could be the case in a program. (Although, there are other ways to deal with that as well. See, for example, Section 4.4 and the function `size()`.)

4.2.2 Addressing using a single index

Row or column vectors can be addressed with just one index that indicates where the subset is located. For example, if `testVector` is the row vector `[1 3 5 7 9 11]`, then the first four elements can be picked out as

```
>> testVector(1:4)
ans =
     1     3     5     7
```

Consequently, you do not have to specify any row number. This works the same way with column vectors. However, matrices can also be addressed using a single index. In that case, the elements are numbered columnwise, starting with the upper-left element. The first seven elements of `testMatrix` are

```
>> testMatrix(1:7)
ans =
     1     2     3    32     4    19     7
```

As the addressing vector, `1:7`, is a row vector, the answer is also a row vector.

Similarly to addressing using row and column numbers, you can pick out all elements using the colon operator alone. Since there is only one dimension when using a single index, all elements are picked out as a vector.

```
>> testMatrix(:)
ans =
     1
     2
     3
    32
     4
    19
     7
     1
     3
     8
     9
     9
```

In this case, the result is a column vector.

4.2.3 Addressing using conditional statements

Matrix subsets can also be picked out based on criteria regarding the elements themselves. Suppose, for instance, that we want to select all elements of `testMatrix` that are greater than 5. Issuing the statement

```
>> index_gt5 = find(testMatrix > 5)
index_gt5 =

     4
     6
     7
    10
    11
    12
```

produces a column vector, `index_gt5`, with a list of which elements in `testMatrix` that are greater than 5. Note that the numbers in this list correspond to addressing the matrix in the single index style (i.e., numbering them columnwise). We can then use the index to pick out the elements in `testMatrix` that are greater than 5.



Strømmen produseres ofte langt fra der den skal brukes.

Statnett sitt oppdrag er å gjøre strømmen tilgjengelig, uansett hvor i dette langstrakte landet du bor. Det er vi som bygger og drifter "riksveiene" i norsk strømforsyning. Gjennom vårt landsdekkende nett sørger vi for en sikker fordeling av strøm mellom nord, sør, øst og vest.

Vi binder Norge sammen

Statnett
Vårt felles kraftnett

Er du student? Les mer her
www.statnett.no/no/Jobb-og-karriere/Studenter



```
>> testMatrix(index_gt5)
ans =

    32
    19
     7
     8
     9
     9
```

Of course, you do not have to take the extra step of defining the index variable. You can do the same thing nesting the commands as `testMatrix(find(testMatrix > 5))`.

There is also an alternative strategy for conditional addressing, using logical variables. If we issue the command

```
>> index_gt5_logic = testMatrix > 5
index_gt5_logic =

     0     1     1     1
     0     0     0     1
     0     1     0     1
```

we get a matrix, `index_gt5_logic`, with only zeros and ones. This matrix is a logical variable (see Section 3.5.2). Each element is an individual answer to the question whether the corresponding element in `testMatrix` is greater than 5 or not, where 1 indicates true (yes, it is greater than 5) and 0 indicates false (no, it is not greater than 5). This type of index can also be used to pick out the values in `testMatrix` that are greater than 5.

```
>> testMatrix(index_gt5_logic)
ans =

    32
    19
     7
     8
     9
     9
```

Alternatively, you can nest the commands as `testMatrix(testMatrix > 5)`, which is somewhat shorter than the example using `find()`.

Here you may note a difference between logical variables and ordinary numerical ones. If we change the variable `index_gt5_logic` into a numerical variable by multiplying it with 1, it is no longer possible to use it for addressing.⁵

```
>> index_gt5_log = index_gt5_logic*1;
>> testMatrix(index_gt5_logic)
??? Subscript indices must either be real positive integers
or logicals.
```

4.3 Changing parts of a matrix

The method of addressing a subset of a matrix can also be used to change it. The way to do that is to address the subset that you want to change, followed by the assignment operator (=) and, lastly, what you want to change the subset to. Suppose we want to change the last two observations for the third location in `testMatrix` from 9 and 9 to 7 and 6. The address of those two elements, using row and column numbers, is rows 2 and 3 and column 4. To change those elements, we assign this part the value of a 2x1 matrix with elements 7 and 6:

```
>> testMatrix(2:3,4) = [7 ; 6]
testMatrix =

     1     32     7     8
     2      4     1     7
     3     19     3     6
```

Naturally, the dimensions of the part you want to change (here, 2x1) must be the same as the dimensions of the new data. Note also that you can do the same thing using the single index addressing method:

```
>> testMatrix(11:12) = [7 ; 6]
testMatrix =

     1     32     7     8
     2      4     1     7
     3     19     3     6
```

You can also use conditional addressing to change data. Suppose you are only interested in observations that are less than 10. Then you can set the rest to NaN (not-a-number) by issuing

```
>> testMatrix(testMatrix >= 10) = NaN
testMatrix =

     1    NaN     7     8
     2      4     1     7
     3    NaN     3     6
```

Conditional addressing is also useful when we want to select a certain period from the data. Suppose we only want observations from day 2 and on. Then we first pick out the correct row numbers from the first column, and then use those to select all columns from the selected rows.

```
>> testMatrix(testMatrix(:,1) >= 2,:)
ans =
```

2	4	1	7
3	NaN	3	6

4.3.1 Reducing and increasing the size of a matrix

Sometimes you will want to delete parts of a matrix (i.e., reduce its size). Since matrices cannot have empty entries, you can only delete full rows or full columns. To delete a row or a column, you assign that part an empty value (i.e., square brackets with nothing in between them). For example, to delete column two of the test matrix, we enter

```
>> testMatrix(:,2) = []
testMatrix =
```

1	7	8
2	1	7
3	3	6

Hva får egentlig en ingeniør- eller teknologistudent for 300 kroner?

- Medlemskap i en aktiv studentorganisasjon – hele studietiden
- 150 tillitsvalgte studenter som ivaretar dine interesser
- Jobbsøkerkurs
- Gratis PC-forsikring og gode bank- og forsikringstilbud
- Teknisk Ukeblad og NITO Refleks
- Møteplasser på web 2.0

Flere medlemsfordeler og innmelding: www.nito.no/student

Alle som studerer på ingeniør-, bioingeniør-, sivilingeniør eller andre teknologistudier (høgskolekandidat, bachelor eller master) kan bli medlem i NITO.

NITO NORGES STØRSTE ORGANISASJON
FOR INGENIØRER OG TEKNOLOGER



The addressing picks out all rows of the second column, and then the columns is assigned an empty value (i.e., it is deleted), leaving `testMatrix` a 3x3 matrix.

Increasing the size of a matrix is done automatically if you add one or several elements outside the existing rows or columns. Then Matlab adds enough rows or columns to fit the new elements, and sets the other elements of the new rows or columns to zero. For example, adding day four in row four and column one, automatically adds a fourth row with the first element equal to 4 and the other ones equal to zero.

```
>> testMatrix(4,1) = 4
testMatrix =

     1     7     8
     2     1     7
     3     3     6
     4     0     0
```

4.4 Some special commands for handling matrices

To handle matrices, it is useful to know the following commands.

<code>find(X>3)</code>	Creates a vector of numbers that indicate which elements in <code>X</code> that are greater than 3. (Note: uses single index addressing.) <code>[I, J] = find(X>3)</code> , where you specify that you want <i>two</i> outputs, creates two vectors. One vector, <code>I</code> , with row numbers and another one, <code>J</code> , with column numbers. This is, consequently, an example of a function that works differently depending on which output you ask for.
<code>size(X)</code>	Creates a 1x2 vector where the first element is the number of rows in <code>X</code> and the second element is the number of columns. This is often used in programs when you do not know how large a certain matrix is.
<code>transpose(X)</code>	This flips the columns and rows of the matrix <code>X</code> so that the first row of <code>X</code> becomes the first column of the new matrix, etc. If <code>X</code> , for example, is a 3x5 matrix, then the new one will be a 5x3 matrix. Note: instead of the command <code>transpose(X)</code> , you can write a single quote after the matrix that is to be transposed: <code>X'</code> . This produces the same result but is much simpler.
<code>sort(X)</code>	<code>X</code> sorted from smallest to largest (columnwise). Each column is sorted separately.
<code>sortrows(X, 3)</code>	<code>X</code> sorted as a group from smallest to largest values in the third column. The third column is sorted and observations in the other columns stick to the corresponding values of that column. Entering a negative sorting column number produces a sorting from largest to smallest instead.

Note that, character strings are also matrices; character matrices. Therefore, many matrix commands, including all the ones above, operate on strings as well. Try, for instance, `transpose(sort('Matlab'))`.

Suppose now that we want to sort the data in `testMatrix` with respect to how much it has rained in the first location (i.e., the second column). We then enter

```
>> sortrows(testMatrix,2)
ans =

     4     0     0
     2     1     7
     3     3     6
     1     7     8
```

Note that column 2 is now in ascending order. The values in columns 1 and 3 still correspond to the same values in column 2 as they did before, though. Suppose, instead, we enter

```
>> sort(testMatrix)
ans =

     1     0     0
     2     1     6
     3     3     7
     4     7     8
```


Then all columns are sorted independently of each other, and the relations to the observations in the other columns are lost.

4.5 The Workspace Browser and the Variable Editor

The Workspace Browser is an interface to the variables currently defined. Through the interface you can create, view, or change variables, as well as create graphics using these variables.

If you do not have the Workspace Browser open, you can open it by issuing the command `workspace` from the Command Window or by selecting `Desktop > Workspace` from the drop-down menus.

To view or change a variable, double-click its name in the Workspace Browser. This opens the variable in the Variable Editor, as in Figure 4-1 where we have opened `testMatrix`. As you see in the figure, the elements of the four rows and three columns of `testMatrix` are displayed in spreadsheet fashion, similar to, for instance, Excel.

To change an element, you click the cell, enter a new value, and press <Enter>. If you enter values outside the defined range, Matlab will automatically pad the empty rows and columns needed to include the new cell with zeros. To insert rows or columns between existing ones, select the row or column where you want to add one and type <Ctrl>+;. To delete a row or column, select it and type <Ctrl>-;. Try, for example, deleting all values in row 4. You close the Variable Editor by clicking  in the upper-right corner. Issue `testMatrix` to see that the matrix has changed.



Skatteetaten



Vil du jobbe i et av landets største IT-miljøer?

Vi skal gjøre det kompliserte enkelt

Skatteetaten tilbyr store fagmiljø og utfordrende oppgaver innen:

- > Systemutvikling
- > Service oriented architecture (SOA)
- > Business intelligence (BI)
- > Testledelse
- > Webutvikling
- > IT sikkerhet
- > Infrastruktur
- > Brukergrensesnitt

For nyutdannede IT-spesialister kan vi tilby et to-årig traineeprogram.

Profesjonell • Nytenkende • Imøtekommende

For mer informasjon se skatteetaten.no/jobb



Figure 4-1: The Variable Editor

Variable Editor - testMatrix

File Edit View Graphics Debug Desktop Window Help

Stack: Ba... No valid plots for: testMatrix...

testMatrix <4x3 double>

	1	2	3	4	5	6	7
1	1	7	8				
2	2	1	7				
3	3	3	6				
4	4	0	0				
5							
6							
7							
8							
9							
10							
11							
12							
13							

4.6 More about matrices

- Matlab also supports a concept called *sparse matrices*. Large matrices can consume a lot of memory and computational power. At the same time, many matrices contain a large number of zeros. Using commands that are tailor made for sparse matrices, is a way to use the fact that many elements are zero to make memory usage and computations more efficient.

5 Mathematical operations with matrices

There are different types of mathematical functions that operate on matrices. Here, we will describe three. First, there are functions that operate element-by-element in a way similar to ordinary algebra. Second, there are functions that, while they operate on the whole matrix or a subset of it, are very similar to ordinary algebra, for example functions for summing rows or columns. Third, there are functions that operate on the whole matrix, for example matrix multiplication.

5.1 Functions that operate element-by-element

Most ordinary mathematical functions in Matlab operate element-by-element when used on matrices. The main exceptions are the operators for multiplication, division, and exponentiation. These reason why these do not operate element-by-element is that the standard in Matlab is matrix algebra. And matrix multiplication, exponentiation, and division (or, rather, inversion) do not operate element-by-element. Addition and subtraction, however, operate element-by-element in matrix algebra as well, so in those cases there are no differences. All four basic arithmetic operations are described in Section 5.3, where the topic is matrix algebra.

Informally, ordinary mathematical functions that take the arguments within parentheses, such as `sqrt()` and `log()`, operate element-by-element, whereas functions that do not use parentheses, such as `*` or `/`, operate according to matrix algebra.

To multiply or divide matrices element-by-element using the ordinary operators `*` and `/`, you have to use so called dot-notation. This means that you put a dot in front of the operators. For example

```
>> X = [2 4 ; 3 4]; Y = [2 2 ; 3 1]; X.*Y, X./Y
ans =
     4     8
     9     4

ans =
     1     2
     1     4
```

Each element in the first answer is the product of the corresponding elements in `X` and `Y`, and each element in the second answer is the corresponding element in `X` divided by its counterpart in `Y`. Note that, the matrices must have exactly the same dimensions. The exception is if either `X` or `Y` is a scalar.

There are actually alternatives to these functions, and to plus and minus as well, that operate element-by-element as well. These are `plus()`, `minus()`, `times()`, and `rdivide()` for, in turn, element-by-element addition, subtraction, multiplication, and division. `rdivide` stands for “right division” (i.e., divide by the argument to the right). There is a “left division” command as well: `ldivide()`.

Other mathematical functions we have seen that use parentheses, such as the square root, also operate element-by-element.

```
>> sqrt(X)
ans =
    1.4142    2
    1.7321    2
```

In Section 4.2.3 we saw that relational operators also work element-by-element when comparing a matrix to a scalar. An example comparing two full matrices instead is

```
>> X > Y
ans =
    0    1
    0    1
```

Each element in `X` is compared to the corresponding element in `Y`. Two of them are not greater than the ones they are compared to; two are.



OLJE- OG ENERGIDEPARTEMENTET



Er du full av energi?

Olje- og energidepartementets hovedoppgave er å tilrettelegge for en samordnet og helhetlig energipolitikk. Vårt overordnede mål er å sikre høy verdiskapning gjennom effektiv og miljøvennlig forvaltning av energiresursene.

Vi vet at den viktigste kilden til læring etter studiene er arbeidssituasjonen. Hos oss får du:

- Innsikt i olje- og energisektoren og dens økende betydning for norsk økonomi
- Utforme fremtidens energipolitikk
- Se det politiske systemet fra innsiden
- Høy kompetanse på et saksfelt, men også et unikt overblikk over den generelle samfunnsutviklingen
- Raskt ansvar for store og utfordrende oppgaver
- Mulighet til å arbeide med internasjonale spørsmål i en næring der Norge er en betydelig aktør

Vi rekrutterer sivil- og samfunnsøkonomer, jurister og samfunnsvitere fra universiteter og høyskoler.

www.regjeringen.no/oed



Se ledige stillinger her

www.jobb.dep.no/oed



5.2 Elementary mathematical functions that operate columnwise

Matrices often contain data observations and it is practical to calculate ordinary summary statistics directly on them. The following are some elementary mathematical and statistical functions that operate on whole columns. For the statistical functions, think of X as a matrix containing many observations of a few variables, where each column represents a certain variable, for example weight or length, and each row contains one observation of each variable.

<code>min(X)</code>	The minimum value of X (columnwise). <code>[val, row]=min(X)</code> produces two vectors. <code>val</code> contains the maximum values for each column and <code>row</code> contains the row numbers where each corresponding maximum is located.
<code>max(X)</code>	The maximum value of X (columnwise). <code>[val, row]=max(X)</code> works similarly to <code>min(X)</code> .
<code>mean(X)</code>	The mean value of each column of X .
<code>median(X)</code>	The median value of each column of X .
<code>std(X)</code>	The standard deviation of each column of X .
<code>var(X)</code>	The variance of each column of X .
<code>sum(X)</code>	The columnwise sum of all values in X .
<code>cumsum(X)</code>	The columnwise cumulative sum of the values in X .
<code>prod(X)</code>	The columnwise product of all values in X .
<code>cumprod(X)</code>	The columnwise cumulative product of the values in X .
<code>diff(X)</code>	The difference between each consecutive element in X , columnwise.

In addition, the following calculate covariances and correlation coefficients.

<code>cov(X)</code>	Calculates the covariance matrix, assuming that each column in X represents outcomes of a variable.
<code>corrcoef(X)</code>	Calculates a matrix of correlation coefficients, assuming that each column in X represents outcomes of a variable.

If, for example, we want to calculate the mean amount of rain for the two remaining locations in `testMatrix`, we enter

```
>> mean(testMatrix(:,2:end))
ans =
    3.6667    7
```

Here, we pick out all rows and columns 2 and 3 and calculate the means of those two columns. Since the first column only contains day numbers, it makes little sense to calculate the mean of that column.

If you want the commands to operate rowwise, instead of columnwise, a convenient way of doing that is to use the transpose operator twice, as in

```
>> mean(testMatrix(:,2:end)')
ans =

    7.5
     4
    4.5
```

This calculates the daily mean amount of rain in the two locations.

Note that, if X is a row vector, the commands here do not operate on the columns of X , but on the single row. For example

```
>> max([1 2 3 4 5 6])
ans =

     6
```

5.3 Matrix algebra

If you have not seen matrix algebra before, it probably seems counterintuitive. It has many uses, though. For instance to solve systems of linear equations and to estimate regression coefficients, as we will see in Section 5.4 and Section 5.5. Here, we only present definitions of basic algebraic operators and how to use these operators in Matlab.

5.3.1 Matrix addition and subtraction

To add or subtract two matrices, they have to be of the same dimensions and all operations are element-by-element. The definition for matrix addition is⁶

$$X + Y = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{T,1} & x_{T,2} & \cdots & x_{T,n} \end{bmatrix} + \begin{bmatrix} y_{1,1} & y_{1,2} & \cdots & y_{1,n} \\ y_{2,1} & y_{2,2} & \cdots & y_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{T,1} & y_{T,2} & \cdots & y_{T,n} \end{bmatrix} = \begin{bmatrix} x_{1,1} + y_{1,1} & x_{1,2} + y_{1,2} & \cdots & x_{1,n} + y_{1,n} \\ x_{2,1} + y_{2,1} & x_{2,2} + y_{2,2} & \cdots & x_{2,n} + y_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{T,1} + y_{T,1} & x_{T,2} + y_{T,2} & \cdots & x_{T,n} + y_{T,n} \end{bmatrix},$$

where both X and Y are $T \times n$ matrices (i.e., they have T rows and n columns). Subtraction is defined analogously. In Matlab, you issue the ordinary addition or subtraction operators, $+$ or $-$. For example,

```
>> X=[1 2 ; 3 4]; Y = [4 3 ; 2 1]; X+Y, Y-X
ans =
     5     5
     5     5

ans =
     3     1
    -1    -3
```

Note that you can add or subtract a scalar from a matrix of any dimensions. The scalar is then added to, or subtracted from, each element in the matrix.

5.3.2 Matrix multiplication

To multiply two matrices, the number of columns of the first matrix must equal the number of rows of the second. If X is a $T \times k$ matrix and Y is a $k \times n$ matrix, then $X*Y$ is a $T \times n$ matrix (i.e., the product has the same number of rows as the first matrix and the same number of columns as the second). The definition is



HELT GRATIS!

S for Skikk & Bank

En bok om ting som er greit å vite når du har flyttet hjemmefra.

dnb.no

DNB

Bank fra A til Å



$$\begin{aligned}
 \mathbf{X} * \mathbf{Y} &= \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,k} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ x_{T,1} & x_{T,2} & \cdots & x_{T,k} \end{bmatrix} * \begin{bmatrix} y_{1,1} & y_{1,2} & \cdots & y_{1,n} \\ y_{2,1} & y_{2,2} & \cdots & y_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{k,1} & y_{k,2} & \cdots & y_{k,n} \end{bmatrix} \\
 &= \begin{bmatrix} x_{1,1} * y_{1,1} + x_{1,2} * y_{2,1} + \cdots + x_{1,k} * y_{k,1} & x_{1,1} * y_{1,2} + x_{1,2} * y_{2,2} + \cdots + x_{1,k} * y_{k,2} & \cdots & x_{1,1} * y_{1,n} + x_{1,2} * y_{2,n} + \cdots + x_{1,k} * y_{k,n} \\ x_{2,1} * y_{1,1} + x_{2,2} * y_{2,1} + \cdots + x_{2,k} * y_{k,1} & x_{2,1} * y_{1,2} + x_{2,2} * y_{2,2} + \cdots + x_{2,k} * y_{k,2} & \cdots & x_{2,1} * y_{1,n} + x_{2,2} * y_{2,n} + \cdots + x_{2,k} * y_{k,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{T,1} * y_{1,1} + x_{T,2} * y_{2,1} + \cdots + x_{T,k} * y_{k,1} & x_{T,1} * y_{1,2} + x_{T,2} * y_{2,2} + \cdots + x_{T,k} * y_{k,2} & \cdots & x_{T,1} * y_{1,n} + x_{T,2} * y_{2,n} + \cdots + x_{T,k} * y_{k,n} \end{bmatrix}
 \end{aligned}$$

Note that the definition implies that, in general, $\mathbf{X} * \mathbf{Y} \neq \mathbf{Y} * \mathbf{X}$, and that for it to be possible to calculate both $\mathbf{X} * \mathbf{Y}$ and $\mathbf{Y} * \mathbf{X}$, both matrices have to be square matrices.

In Matlab, you calculate matrix multiplication by using the ordinary multiplication operator, `*`. For example, using the previously defined `X` and `Y`

```

>> X*Y, Y*X
ans =
     8     5
    20    13

ans =
    13    20
     5     8

```

As you see from the example, multiplying `X` by `Y` from the left or from the right, premultiplying or postmultiplying, yields different answers. As an alternative to using the operator `*`, you can multiply matrices using the command `mtimes(X,Y)`.

5.3.3 Inverting a matrix

Division is not defined in matrix algebra. However, there is a corresponding concept that applies to square matrices. Remember that, dividing two scalars is equivalent to multiplying the first scalar with the inverse of the second: $\frac{x}{y} = x * \frac{1}{y}$. As inversion is defined in matrix algebra, multiplying a matrix with the inverse of another is similar to dividing the two matrices.

To understand the definition of matrix inversion, note that for scalars $x * \frac{1}{x} = 1$. An informal implicit definition of scalar inversion could then be “the number you multiply the scalar with to get the answer 1”. What is the matrix equivalent of the number 1? The so-called identity matrix, usually denoted \mathbf{I} , is a square matrix with ones on the diagonal, and zeros elsewhere. This, you might recall, is the matrix that you create with the command `eye()`.⁷ If you multiply any square matrix with \mathbf{I} , the result is the matrix you started with, so it is similar to multiplying a scalar with 1.

The identity matrix is used in the definition of matrix inverse. If there exist two matrices A and B such that $A*B = I$, then B is the inverse of A and is denoted A^{-1} . It can be shown that if $A*B = I$ then it is also the case that $B*A = I$, so in this case it does not matter if you multiply from the right or from the left. Note, however, that not all square matrices are invertible, just as it is not possible to invert the scalar 0.

In Matlab, you calculate matrix inverse with `inv()` or by raising the matrix to the power of -1 .

```
>> inv(X), X^-1
ans =
      -2      1
      1.5     -0.5
ans =
      -2      1
      1.5     -0.5
```

Matlab, however, also allows for matrix inversion using the division operators, `/` and `\`, although this notation is nonstandard. In these cases, the matrix that is *above* the division sign (i.e., to the left of `/` or to the right of `\`) is multiplied with the inverse of the other matrix. For example,

```
>> eye(2)/X, X\eye(2)
ans =
      -2      1
      1.5     -0.5
ans =
      -2      1
      1.5     -0.5
```

give the same results as above, and consequently also invert X . Furthermore

```
>> Y/X, Y*inv(X)
ans =
     -3.5     2.5
     -2.5     1.5
ans =
     -3.5     2.5
     -2.5     1.5
```

are two different ways of calculating $Y*X^{-1}$. As alternatives to using the operators `/` and `\`, you can use the commands `mrdivide(X,Y)` and `mldivide(X,Y)`, where the former is matrix right division and the latter is matrix left division.

5.3.4 Commands for linear algebra

Commonly used commands for linear algebra include

<code>norm(X)</code>	Matrix or vector norm.
<code>rank(X)</code>	Matrix rank.
<code>det(X)</code>	Determinant.
<code>trace(X)</code>	The sum of the diagonal elements.
<code>chol(X)</code>	Cholesky factorization.
<code>eig(X)</code>	Eigenvalues and eigenvectors.
<code>expm(X)</code>	Matrix exponential.
<code>logm(X)</code>	Matrix logarithm.
<code>sqrtm(X)</code>	Matrix square root.


5.4 Solving systems of linear equations

Matrix algebra provides an easy way to solve systems of linear equations. Suppose we have the following system of three equations

$$\begin{cases} x_1 + 2x_2 + 3x_3 = 15 \\ x_1 + 2x_2 + x_3 = 13 \\ -2x_1 + 5x_2 - 2x_3 = 19 \end{cases}$$

WANT TO GET TO KNOW BERLIN?

We might have a job for you! Zalando has quickly become Europe's largest online fashion retailer, operating from the heart of Berlin. Want to join our international team?
www.zalando.no/jobb-hos-zalando

 zalando



Such a system can be written in matrix notation as $A^*X = Y$, where

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 1 \\ -2 & 5 & -2 \end{bmatrix}, X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \text{ and } Y = \begin{bmatrix} 15 \\ 13 \\ 19 \end{bmatrix}$$

Now, if we premultiply each side of the equality by the inverse of A, we get

$$A^{-1} * A * X = A^{-1} * Y$$

$$I * X = A^{-1} * Y$$

$$X = A^{-1} * Y$$

To calculate the solution in Matlab, we use either the `inv()` command or `\`.

```
>> A=[1 2 3 ; 1 2 1 ; -2 5 -2]; Y=[15 13 19]'; X=A\Y, X=inv(A)*Y
```

```
X =
```

```
2
```

```
5
```

```
1
```

```
X =
```

```
2
```

```
5
```

```
1
```

Consequently, $x_1 = 2$, $x_2 = 5$, and $x_3 = 1$. To check this, issue

```
>> A*X
```

```
ans =
```

```
15
```

```
13
```

```
19
```

which equals Y .

5.5 Finding linear regression coefficients

You can find linear regression coefficients in a way similar to how we solved the system of linear equations. Suppose we have a large number of observations, say n observations, for one or several independent variables, say p variables, and equally many observations of a dependent variable, and that the model is

$$y_i = \beta_0 + \beta_1 x_{i,1} + \cdots + \beta_p x_{i,p} + \varepsilon_i, i = 1, \dots, n.$$

Then we can write this in matrix form as $\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}$, where

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \mathbf{X} = \begin{bmatrix} 1 & x_{1,1} & \cdots & x_{1,p} \\ 1 & x_{2,1} & \cdots & x_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n,1} & \cdots & x_{n,p} \end{bmatrix}, \boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{bmatrix}, \boldsymbol{\varepsilon} = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{bmatrix}.$$

In other words, \mathbf{y} contains all observations of the dependent variable, \mathbf{X} contains all observations of the p independent variables preceded by a column of ones to match the constant term (β_0), $\boldsymbol{\beta}$ contains the $p+1$ coefficients, and $\boldsymbol{\varepsilon}$ contains the error terms. Note that, if you perform the matrix multiplication, you get the same as in the preceding expression.

There are several ways to estimate $\boldsymbol{\beta}$, but the most frequently used is the OLS estimate (Ordinary Least Squares). In matrix algebra, the estimator can be written: $\hat{\boldsymbol{\beta}} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$, where the “hat” on $\boldsymbol{\beta}$ indicates that it is an estimate.

You can use the OLS formula in Matlab by issuing `inv(X'*X)*X'*y` or `(X'*X)\X'*y`, but there is actually a better way. Looking at the matrix version of the model again, we see that if we could have premultiplied \mathbf{y} with the inverse of \mathbf{X} , and ignore the error terms, this would have solved for $\boldsymbol{\beta}$. Mathematically, this is not correct, since \mathbf{X} is generally not invertible. However, Matlab supports this notation for solving linear regressions, so you can calculate the OLS estimate of $\boldsymbol{\beta}$ as `X\y`. Internally, Matlab uses a completely different method to estimate $\boldsymbol{\beta}$ in this latter case, though. One that is more efficient. So using `X\y` is the preferred method in Matlab. To illustrate this, let us create sample variables and perform the regression.

```
>> X=[ones(1000,1) randn(1000,3) ]; e=randn(1000,1)*0.2;
>> beta=[0.1 0.2 0.3 0.4]'; y=X*beta+e;
>> betahat_1=inv(X'*X)*X'*y, betahat_2=X\y
betahat_1 =
    0.10063
    0.2065
    0.30098
    0.41276
betahat_2 =
    0.10063
    0.2065
    0.30098
    0.41276
```

The first two lines create the X , ε , β , and y matrices. X is a matrix of random numbers and a column of ones, e is a vector of random numbers that are scaled to reduce the noise, β is a column vector of coefficients, and y is a vector created using the other variables. The estimators, `betahat_1` and `betahat_2`, both produce the same estimates. The estimates, in turn, are close to the original β .



"I studied English for 16 years but...
...I finally learned to speak it in just six lessons"

Jane, Chinese architect

ENGLISH OUT THERE

Click to hear me talking before and after my unique course download



6 Importing and exporting data

As data sets grow larger, it becomes inconvenient to type them directly into the program, and for larger data sets, this is close to impossible. There are several other ways to import data into Matlab, though. Importing data from an external source can sometimes be a chore, however. There are many different file formats that data can be stored in, and it can be stored in different ways within files. In addition, data can be more or less well behaved within a file, for example be stored in the same way, or not, throughout the file. In short, the more the data looks like a matrix, the more easily it will be imported. There are, consequently, many ways in which importing can go wrong, and this makes the topic of importing data complex.

Here, we limit the description to some of the most frequently used ways of importing data. Most common types of data are either stored as plain text or as spreadsheets, and we focus on importing from white-space delimited text data and importing from Excel data sheets. Exporting, on the other hand, is usually easier and we describe only how to export data to Excel.

There are three main ways to import data. Either you do it manually, by copying-and-pasting or by using a tool called the Import Wizard, or you do it using a command, which allows you to automate the importing process.

Throughout this section, remember that Matlab can be fussy when it comes to importing data. Usually the data has to be reasonably “well behaved”. For example, if you want to import data from a text file, the data should be organized in columns and rows, and there must be equally many columns in each row. Preferably, there should be no text at all in the file you are importing from, if you are importing numerical data. Note, however, that it is possible to use Matlab comment-style within the text files from which you import. So if you want to keep text within the file, add a %-character at the beginning, and then the rest of that row will be ignored by Matlab. This is a good way to keep comments about what the data is used for or variable labels.

6.1 The Current Folder

Remember the Current Folder from Section 2 and Section 3.7. This is the folder that Matlab will read data from, and store data to, if no address is specified. It is convenient to put files that you want to import in the Current Folder. If you have closed the Current Folder browser, you can open it again by selecting `Desktop > Current Folder` from the drop-down menus at the top of the window.

If you are using your own computer, it is good to have a default folder where you keep data and work in progress and choose that as Current Folder. If you use a shared computer, it can be good to use a USB-stick as Current Folder, so that you can easily bring the files and data that you have saved with you after the session.

To select folder, browse to the one you want, either from the address bar at the top of the Current Folder browser, or from the address bar on the Matlab toolbar. If the latter is closed, you open it by selecting Desktop > Toolbars > MATLAB. You can also use commands. If you issue the command `cd` (for *Current Directory*) from the Command Window, Matlab responds by typing the address to the Current Folder. If, instead, you enclose a full address path within parentheses and single quotes after the command, the Current Folder is changed to the new address.

```
>> cd('C:\Users\Public\Documents\'), cd  
C:\Users\Public\Documents
```

6.2 Problems with importing formatted data

Numbers can be formatted in different ways. In North America, the UK, Australia, and large parts of Asia, decimals are indicated with a period/stop and large numbers are often grouped in thousands by separating them with commas. 1000000π could, for example, be written 3,141,592.65 with two decimals. In most of Europe and South America, decimals are instead indicated with a comma and large numbers are often grouped by separating them with either blank space or a period/stop. So 1000000π could be written 3 141 592,65 or 3.141.592,65.

Different data formats can cause problems when importing, particularly if you import from text data or by pasting copied data. In such cases, the formatting must follow the conventions of Matlab. Consequently, decimals must be indicated with a period/stop and large number must not be grouped at all. If your data does not adhere to this convention, you must first change it so that it does, for example by using a find-and-replace tool in a text editor.

In Excel, this is different. Numbers are stored in a certain way internally, but are presented on the screen in a format chosen by the user inside of Excel and/or by the numerical format chosen in the settings for the computer. If you copy data from Excel and paste it to Matlab, the rule is the same as above: The data must follow the conventions of Matlab. However, if you import data from an Excel file, using the Import Wizard or using commands, then Matlab ignores the presentation format and looks directly to how it is stored internally. Then numerical data is recognized regardless of grouping and decimal character conventions. Note, however, that numbers that are stored as text in Excel will be imported as text to Matlab.

6.3 Preparing data to import

We need a sample data set to import. Copy the data below into a text editor, such as Notepad, and save it as `testData_text.txt`.⁸ Then move the file to the Current Folder.

```
1975    15    92
1976   222    90
1977    19    87
1978    73    94
1979   150    92
```

Also create an Excel version of the data, named `testData_Excel.xlsx` (or, if you have an older version of Excel, with the suffix `.xls`), and put that file in the Current Folder as well.

Note that the data is organized in three columns that are separated by white space (one or several blanks) in the text document, and is organized in cells in the Excel document. There is no text in the files and they are easy to interpret as a matrix. This makes this data easy to import.

6.4 Copy-and-paste importing

Copy-and-paste importing is very simple. Open either the text or the Excel document you just created. Select the data and copy it. Then move back to Matlab. In the Command Window, type a variable name, the assignment operator, left and right square brackets and a semicolon (to suppress output), for example `importedData = [];`. Then place the cursor between the square brackets, paste the data, and press <Enter>.

```
>> importedData = [1975 15 92
1976 222 90
1977 19 87
1978 73 94
1979 150 92];
```

Check the Workspace to see that the data has been imported as the new variable `importedData`.

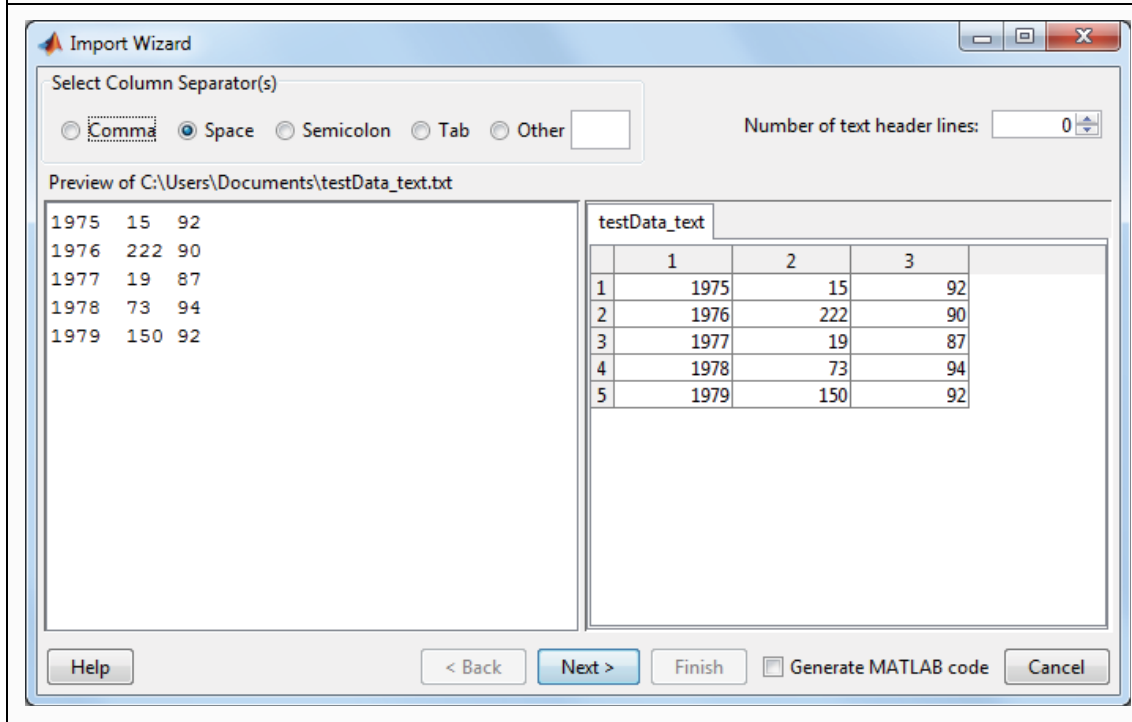
There is also a tool in Matlab for copying-and-pasting data. After you have copied the data you want to import, go to Matlab, and choose `Edit > Paste to Workspace...`. This opens up the Import Wizard that is described in the next section, and you proceed in the same way as there, except that you do not have to specify a file from which to import.

6.5 Importing using the Import Wizard

The Import Wizard is very useful when you are not completely sure how Matlab reads the data, since an interpretation is presented in the Wizard and you can choose different ways to import.

Open the Wizard by choosing `File > Import Data...`, or by issuing `uiimport` from the Command Window and then clicking "File" in the dialogue. Select the file you want to import, `testData.txt`, and click "Open". This opens the Wizard with the selected file in a new window (see Figure 6-1).

Figure 6-1: The Import Wizard; Importing from a text file, step 1



WHILE YOU WERE SLEEPING...

DUKE
THE FUQUA
SCHOOL
OF BUSINESS

www.fuqua.duke.edu/whileyouweresleeping



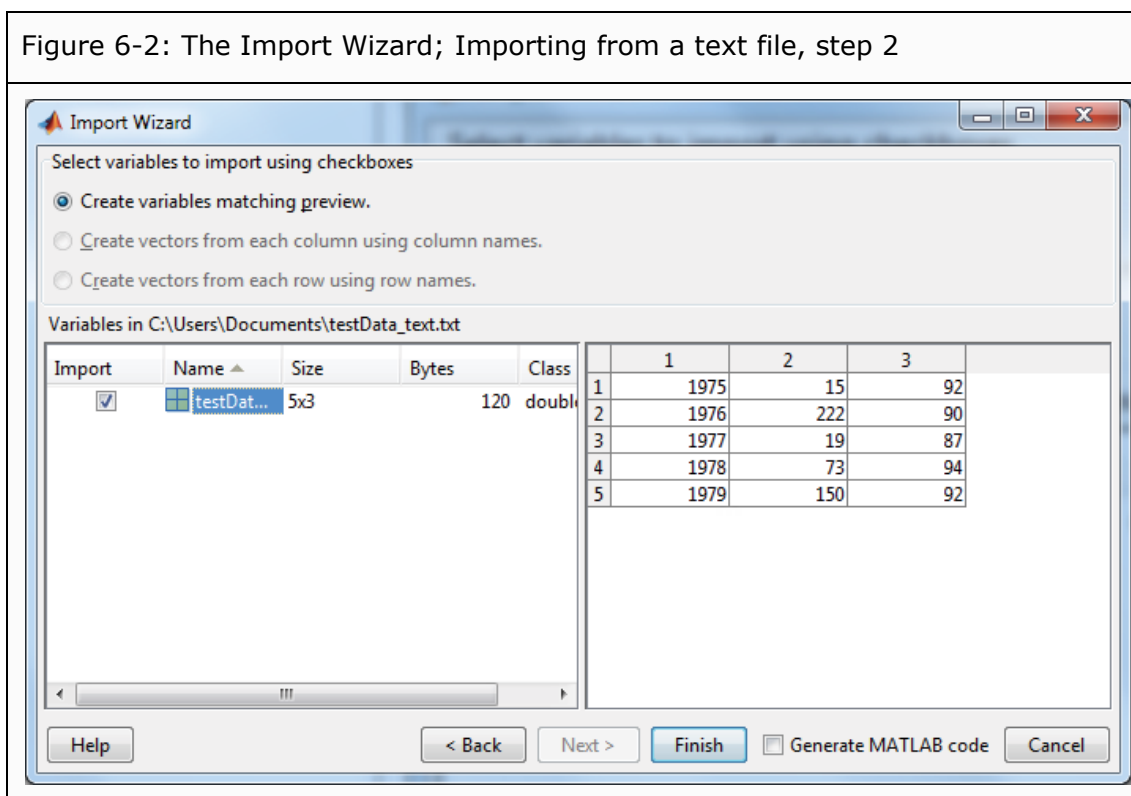
At the top of the new window, you can select which character that separates the data. Matlab preselects the one that seems most likely, "Space" in our case. In the left part of the window, the data is shown as it looks inside the chosen file. In the right part, the data is shown as interpreted by Matlab, given the separator that has been chosen. If you choose some other separator at the top, the contents of the right part will change.

Note the "Number of text header lines" in the upper right part of the window. The Import Wizard allows for a number of text lines at the beginning of the file. If your data has text at the beginning, the Wizard will create two additional variables with default names "textdata" and "colheaders". The first of these will contain all the text at the beginning of the file and the second will contain only the last row before the data begins, interpreted as data labels.

The easiest way to deal with this is usually to not have any text in the data file. Note that, if the data has text within the numerical data, the Wizard will stop importing on the line where the text is located, and ignore the rest. It is therefore important to check files that you import for any anomalies. Again, text that is preceded by a %-character is ignored, even at the beginning of the file.

The test data contains no text and so is easy to interpret. To proceed, click "Next".

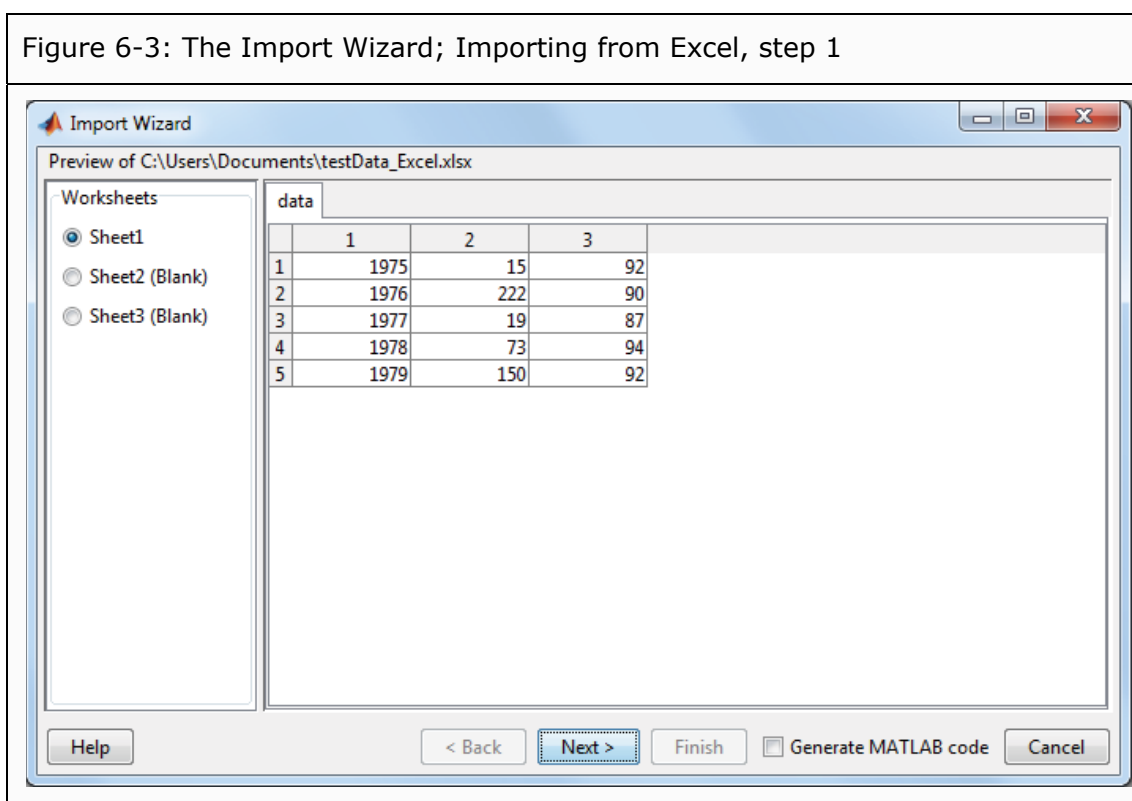
Figure 6-2: The Import Wizard; Importing from a text file, step 2



In the next window (see Figure 6-2), Matlab presents information about the variable that is going to be imported. The name is taken from the name of the file and you can see that the size of the variable, in this case, is a 5x3 matrix. If you select the variable name by clicking it, a preview, similar to the one in the previous window, appears to the right. You can also change the name by clicking it after it has been selected, and typing an alternative. Click "Finish" to conclude the importing. Verify in the Workspace that the variable `testData_text` has been created.

Using the Import Wizard on Excel data is very similar. In Figure 6-3 you see the first step, where you select which sheet in Excel to import from in the panel on the left. The second step is then identical to the second step when importing from a text file.

Figure 6-3: The Import Wizard; Importing from Excel, step 1



6.6 Importing using commands

6.6.1 Importing from text files with commands

Manually importing data by copy-and-paste or using the Import Wizard becomes tedious if you repeatedly import the same file, or the same type of files. It is then much more convenient to use commands to import data. If importing data is part of a program, you should usually do it using a command from within the program.

There is, however, a caveat. When you import data using the Wizard, you get a preview and it is often easy to spot data problems. When it comes to importing using a command, the data must be even more “well behaved” for the importing to work. For instance, problems can arise if there are missing observations within the data, or if there are any nonstandard numerical formats in text files. It is not always that Matlab issues a warning when data is not properly imported.

If, however, the data is well behaved, importing text files is very easy. Suppose you want to import the data in `testData_text.txt` as a new variable named `myData`. You then enter

```
>> myData = load('testData_text.txt');
```

The new variable `myData` then appears in the Workspace. Note that the file name has to be enclosed within single quotes and that the file extension, `.txt`, must be included. Enter the variable name in the Command Window to view the contents.

```
myData =
    1975         15         92
    1976        222         90
    1977         19         87
    1978         73         94
    1979        150         92
```



Vi vokser i Norge
og har virksomhet
helt frem til 2050

Er du interessert i sommerjobb
eller fast stilling?

Se informasjon om sommerjobber på
www.bp.no

bp



6.6.2 Importing from Excel files with commands

Importing Excel files is slightly more complicated, since this kind of file is more complex. An Excel workbook might mix numerical and non-numerical data, and can have several different sheets of data. Since the data in our example file, `testData_Excel.xlsx`, is very simple, it is possible to import it by issuing the Excel-read command, `xlsread()`, similarly to the example with the text file above.

```
>> myExcelData = xlsread('testData_Excel.xlsx');
```

If no sheet name is presented, Matlab assumes that the first sheet is to be read. And since the data in our example only contains numerical data and is located in the first sheet, it is easily read.

Let us make the example more complex. Open the data in Excel. Then rename the sheet where the data is located "Data". Also, insert data labels on the first row, for example Year, Chicago, and New York, so that the sheet now contains both numerical and non-numerical data.⁹ (You can check how Matlab handles the new situation in the Import Wizard, if you try using it on the changed data set. Similarly to when there were data labels in a text file, Matlab suggests creating three files named "data", "colheaders", and "textdata".)

It might surprise you that, if you issue the same command as above to import the data, you get exactly the same result. But what happened to the new information, the sheet name, and the labels? Since no sheet name was entered in the command, Matlab assumed the first sheet was to be read. And that is exactly where our data is. Furthermore, the sheet mixes numerical and non-numerical data. However, when issued in the way that we have, the `xlsread()` command only imports the numerical data.

To make use of the new information, issue instead

```
>> [myExcelData, labels] = xlsread ('testData_Excel.xlsx', 'Data')
myExcelData =
```

1975	15	92
1976	222	90
1977	19	87
1978	73	94
1979	150	92

```
labels =
    'Year' 'Chicago' 'New York'
```

Many commands in Matlab can return more than one output. In this case, `xlsread()` has two output variables, `myExcelData` and `labels`. In this case, numerical data is imported to the first variable and non-numerical data to the second.¹⁰

You can also choose to import only part of an Excel sheet by adding a third input argument that specifies the upper-left and lower-right corner of the rectangular area of the sheet from which you import. For example

```
>> myData = xlsread('testData_Excel.xlsx', 'Data', 'B2:C4')
myData =

    15    92
   222    90
    20    87
```

See `help xlsread` for many more examples and tips.

So far, we have assumed that the data is stored in the Current Folder. What if it is not? In that case, you have to type the whole path to the file. For example, the command could be

```
>> data=xlsread('C:\Users\Public\Documents\testData_Excel.xlsx');
```

6.7 Exporting to Excel files with commands

To export data to Excel, you use the command `xlswrite()`. For instance, you can create a new Excel file by using a new file name and specifying which variable you want to export. The following creates a 2x5 matrix of random data, `random`, and then exports the matrix to an Excel file named `randomExcel.xlsx` in the Current Folder.

```
>> random = randn(2,5);
>> xlswrite('randomExcel.xlsx', random)
```

Check the Current Folder to see that a new Excel document, `randomExcel`, has appeared there. If you open the document, you find the random data. Note that, if a document by the same name already exists in the folder to which you export the data, Matlab will overwrite the existing data. Also, note that, the file to which you are exporting cannot be open in Excel while you export the data.

As when reading from an Excel file, you can also specify a sheet name and a range. For the range, it is enough that you specify the upper-left corner. Then Matlab will automatically calculate the range based on the size of the data you want to export. The following adds a new sheet to the data, named "Random Data", and then saves the same random matrix as before, but starting in column B and row 2 instead. When Matlab creates a new sheet in an Excel file, it issues a warning in the Command Window.

```
>> xlswrite('randomExcel.xlsx', random, 'Random Data', 'B2')  
Warning: Added specified worksheet.
```

6.7.1 Saving and loading Matlab variables

When you have imported data into Matlab, you have it stored locally as Matlab variables (i.e., the variables listed in the Workspace). When you exit Matlab, all those variables will be lost. But you can, and often should, save them for future use.

To save all variables in the Workspace, type `save` and a name for the variable collection

```
>> save myVariables
```

There is no confirmation that the variables were actually saved. To see that they were, delete the variables, using `clear`, and then load them again using the command `load` and the name you used when saving them.

```
>> clear  
>> load myVariables
```

Check the Workspace to see that all variables are back again.

To save only a subset of the variables, type `save`, a name for the variable collection, and then a list of all the variables you want to save

```
>> save mySelectedVariables testData_text labels myData
```

Again, to see that they have been saved you can delete them, using `clear` and the list of variables (which only deletes the variables in the list, leaving all others), and then loading them back again.

```
>> clear testData_text labels myData  
>> load mySelectedVariables
```

The variables `testData_text`, `labels` and `myData` should disappear from the Workspace after the first command, and then reappear after the second. The variables are, by default, saved to files in the Current Folder. If you check, you will now find two files with the names `myVariables` and `mySelectedVariables` there.

6.8 More about importing and exporting data

- For files of comma-separated-values (`.csv`), see commands `csvread` and `csvwrite`.
- For text files with other delimiters than white space, see commands `dlmread` and `dlmwrite`.
- Matlab can also import audio, image, and video formats, as well as some different types of specialized scientific data formats. See `help fileformats` for a list of formats.

7 Graphics

There are numerous ways to produce graphs in Matlab, as well as an enormous amount of tweaks to make the graphs appealing, and we will only provide a short introduction here on how to produce the most widely used types of graphs.

To begin, we need some data to work with. For simplicity, let us use some random data. We have already seen commands that create matrices of random data as well as commands that calculate cumulative sums and products on matrices (see sections 4.1.1 and 5.2). We can use these commands to create a series that moves randomly up and down.

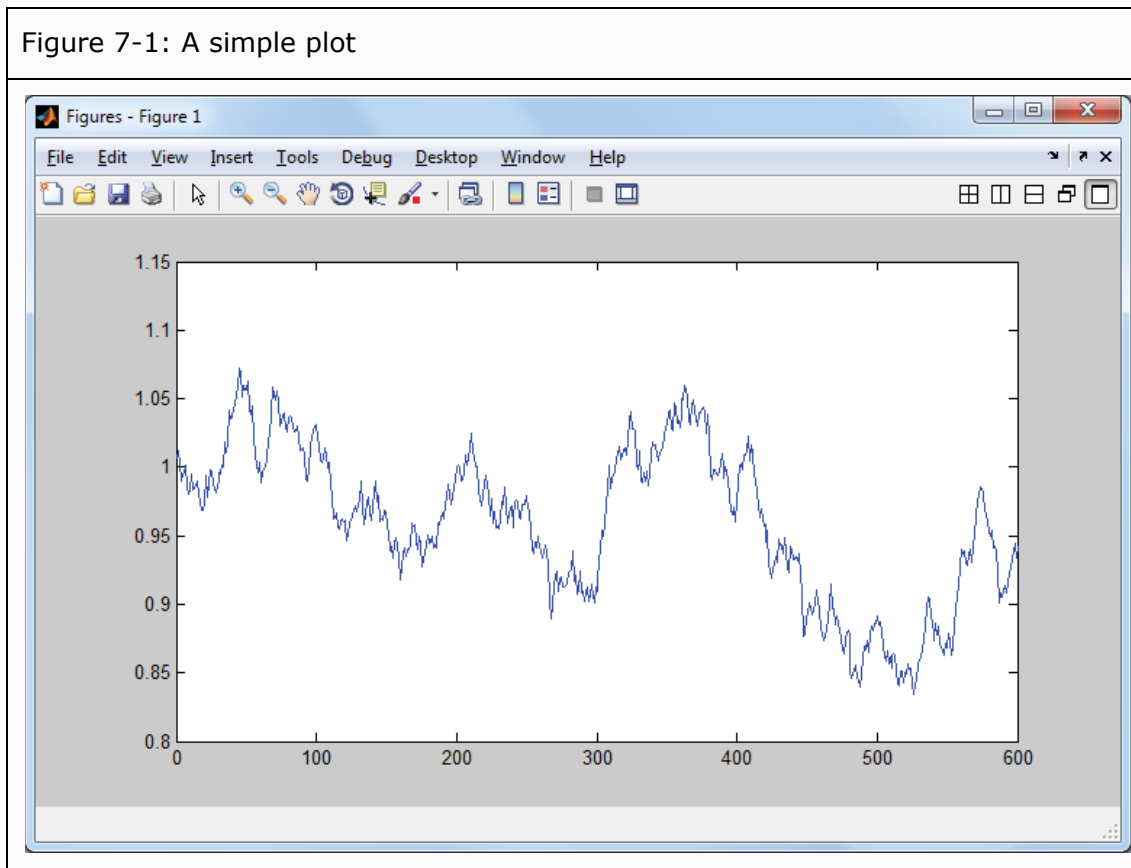
```
>> obs = cumprod(1+randn(600,1)/100);
```


This line, that nests several functions, first creates a 600x1 matrix of random numbers (from a normal distribution), divides them all by 100, adds 1, and then, finally, multiplies them cumulatively. The easiest way to produce a plot of a variable is to issue the `plot()` command.

```
>> plot(obs)
```



This produces a plot of the variable `obs` (see Figure 7-1). Since we have not defined any X-variable, the observation numbers are used for the X-scale. If there is no figure window open, one is automatically opened after the `plot()` command is issued.¹¹



The new window will not be docked to the Desktop. As explained in Section 2, you can dock it by clicking , and you can move it around the Desktop by clicking-and-dragging the name list at the top of the figure window. The size of the axes is calculated by Matlab to make the graph look reasonably good. This often leaves a little space to the left and to the right, as well as at the top and at the bottom, of the graph. However, when values seem to be bounded by zero, or some other number that makes for a nice looking axis, Matlab has a preference to begin and end the corresponding axis there, as is the case in the figure, where there is only extra space at the top and at the bottom.

7.1 Useful commands for two-dimensional plotting

The most frequently used plotting commands include the following.


<code>plot(x,y)</code>	<p>Produces a two-dimensional plot. <code>x</code> should be a vector of X-values and <code>y</code> should be a vector or a matrix with corresponding Y-values. They must be of equal length. Note that, if there is already a plot in the figure window, it is erased and replaced by the new one.</p> <p>You can plot several lines by entering <code>x</code>- and <code>y</code>-vectors alternatively in the plot command, for example <code>plot(x1,y1,x2,y2,x3,y3)</code>. If the <code>x</code>-vectors are the same, you can enter all <code>y</code>-vectors as a matrix, for example <code>plot(x1,[y1 y2 y3])</code>.</p> <p>You can also add code within single quotes about how you want the data to be presented. For example, <code>plot(x,y,'r:d')</code> produces a red dotted line with diamond markers. If no line type is entered, solid is the default. Codes to enter include:</p> <p><i>colors:</i> <code>b</code> blue; <code>g</code> green; <code>r</code> red; <code>c</code> cyan; <code>m</code> magenta; <code>y</code> yellow; <code>k</code> black; <code>w</code> white</p> <p><i>lines:</i> <code>-</code> solid; <code>:-</code> dotted; <code>-.</code> dashdot; <code>--</code> dashed</p> <p><i>markers:</i> <code>.</code> point; <code>o</code> circle; <code>x</code> x-mark; <code>+</code> plus; <code>*</code> star; <code>s</code> square; <code>d</code> diamond; <code>v</code> triangle; <code>^</code> triangle; <code><</code> triangle; <code>></code> triangle; <code>p</code> pentagram; <code>h</code> hexagram</p> <p>See <code>help plot</code> for more details.</p>
<code>hold</code>	<p>Keeps an existing plot in the figure window when a new one is plotted. <code>hold on</code> turns the feature on and <code>hold off</code> turns it off. <code>hold</code> toggles between on and off.</p>
<code>grid</code>	<p>Adds a grid to the graph. <code>grid on</code> turns the grid on and <code>grid off</code> turns it off. <code>grid</code> toggles between on and off.</p>
<code>xlabel,ylabel</code>	<p>Adds a label to the X- and Y-axis, respectively. For example, <code>xlabel('year')</code> makes the text "year" appear below the X-axis.</p>
<code>title</code>	<p>Adds a name at the top of the graph. For example, <code>title('Interest rates')</code>.</p>
<code>legend</code>	<p>Adds a legend to the graph indicating what the lines represent. For example, <code>legend('short rate','long rate')</code></p>
<code>axis</code>	<p>Controls the appearance of the axes. <code>axis([0 10 2 20])</code> makes the X-axis run from 0 to 10 and the Y-axis from 2 to 20. There are also some specialized versions of this command. <code>axis tight</code>, for example, implements the minimum axes that can still represent all data. See <code>help axis</code> for more tips.</p>

<code>xlim, ylim</code>	Similar to <code>axis</code> but operate only on the X- and Y-dimension, respectively.
<code>figure</code>	Opens up a new figure window (so that a new plot will not erase an old one).
<code>subplot</code>	Partitions the figure window into several rows and/or columns and selects which of these to issue plotting commands to. For example, <code>subplot(4,2,7)</code> partitions the figure window into four rows and two columns and selects the seventh subsection. (Note: counting rowwise, not columnwise. The seventh subsection is, consequently, on the fourth row and in the first column.)

7.2 Time series plotting

Let us produce a few better looking graphs, using some of the commands listed. Let us also invent a time axis for the plot. Say the first observation corresponds to January 15, 1950, and that the following observations correspond to the middle of the following months. January 15 is roughly 1/24-th of a year, and then each consecutive month is 1/12-th of a year later. Since we have 600 observations in the variable `obs`, we can create a matrix with time observations as years, with year fractions, using the colon operator. Note that 600 monthly observations correspond to 50 years and that the series therefore should end slightly before the year 2000.¹²

```
>> dates = 1950+1/24 : 1/12 : 2000;
```



gaiteye®
Challenge the way we run

**EXPERIENCE THE POWER OF
FULL ENGAGEMENT...**

**RUN FASTER.
RUN LONGER..
RUN EASIER...**

READ MORE & PRE-ORDER TODAY
WWW.GAITEYE.COM



Alternatively, we can use the `linspace()` function, noting that the first date should be $1950+1/24$, the last should be 599 months later at $1950+1/24+599/12$, and that there should be 600 observations. The following line produces the same row vector of dates as the previous one.

```
>> dates = linspace(1950+1/24,1950+1/24+599/12,600);
```

Now, we can plot our observations against the dates. We use a green solid line with no markers.

```
>> plot(dates,obs,'g')
```

Note that the old plot disappears when we produce a new one. Then we add a figure title, labels on the axes, and a grid to make the graph easier to read.

```
>> title('Levels during 1950 to 2000')
>> xlabel('Year'), ylabel('Level')
>> grid
```

To add a legend, we enter

```
>> legend('First observations')
```

Suppose we want to add another set of observations to the graph. Let us create an alternative set of observations and add it. Similarly to before, we create a series of random numbers.

```
>> obs_2 = cumprod(1+randn(600,1)/100);
```

Usually when we instruct Matlab to draw a new plot, the old one is erased. To avoid that, we instruct Matlab to hold the plot, and then plot the second set of observations as a dotted blue line and update the legend.

```
>> hold on
>> plot(dates,obs_2,'b:')
>> legend('First observations','Second observations')
```

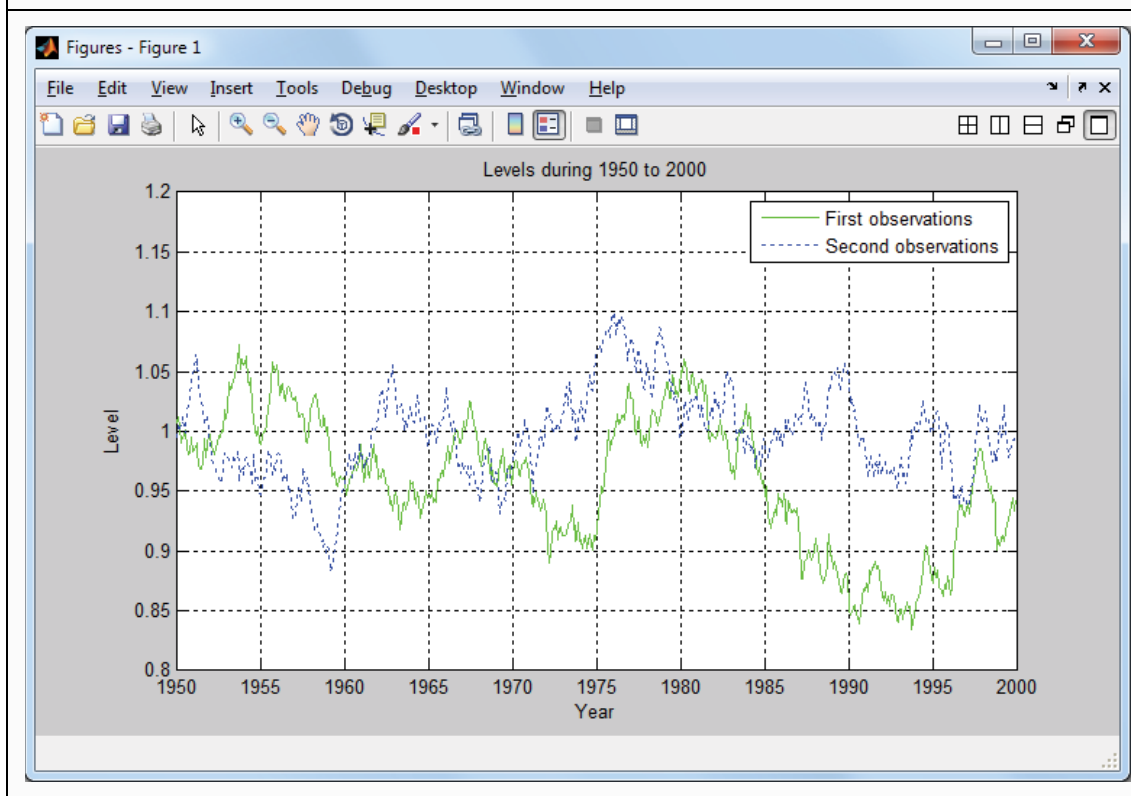
The legend can easily be moved to another location on the graph by clicking and dragging it.

Finally, we adjust the size of the axes, either by using `axis tight` or by explicitly telling Matlab where we want the axes to begin and end.

```
>> axis([1950 2000 0.8 1.2])
```

Figure 7-2 shows the resulting graph. Note that the plot will look different each time, since we are working with random data.

Figure 7-2: A time series plot



7.3 Plotting a function

For a given set of X-values, it is usually easy to create a table of X- and Y-values for a given function, since most functions operate element-by-element. Suppose, for example, that we have the function $y = \sin(x) + e^{-x}$ and want to plot the function for X-values between -1 and 2π . First, we create a vector of X-values, noting that it has to be fine grained enough to represent the changes in Y-values. Then we create a vector of the corresponding Y-values and plot the data (see Figure 7-3). To view the first couple of elements as a look-up table, we display them as a matrix in the Command Window.

```
>> x=linspace(-1,2*pi,100); y=sin(x)+exp(-x); plot(x,y);
>> title('y=sin(x)+exp(-x)'); grid on, axis tight
>> [x(1:5)' y(1:5)']
ans =
        -1          1.8768
    -0.92643        1.726
    -0.85286        1.5932
    -0.7793         1.4772
    -0.70573        1.3767
```

For a given set of X- and Y-values, it is also easy to add markers at specific values, for example at a function minimum. When using the `min()` function with two output arguments, it supplies both the minimum value and the location of that number (see Section 5.2).

```
>> [minVal,minLoc] = min(y)
minVal =
    -0.99091
minLoc =
    79
```



Strømmen produseres ofte langt fra der den skal brukes.

Statnett sitt oppdrag er å gjøre strømmen tilgjengelig, uansett hvor i dette langstrakte landet du bor. Det er vi som bygger og drifter "riksveiene" i norsk strømforsyning. Gjennom vårt landsdekkende nett sørger vi for en sikker fordeling av strøm mellom nord, sør, øst og vest.

Vi binder Norge sammen

Statnett
Vårt felles kraftnett

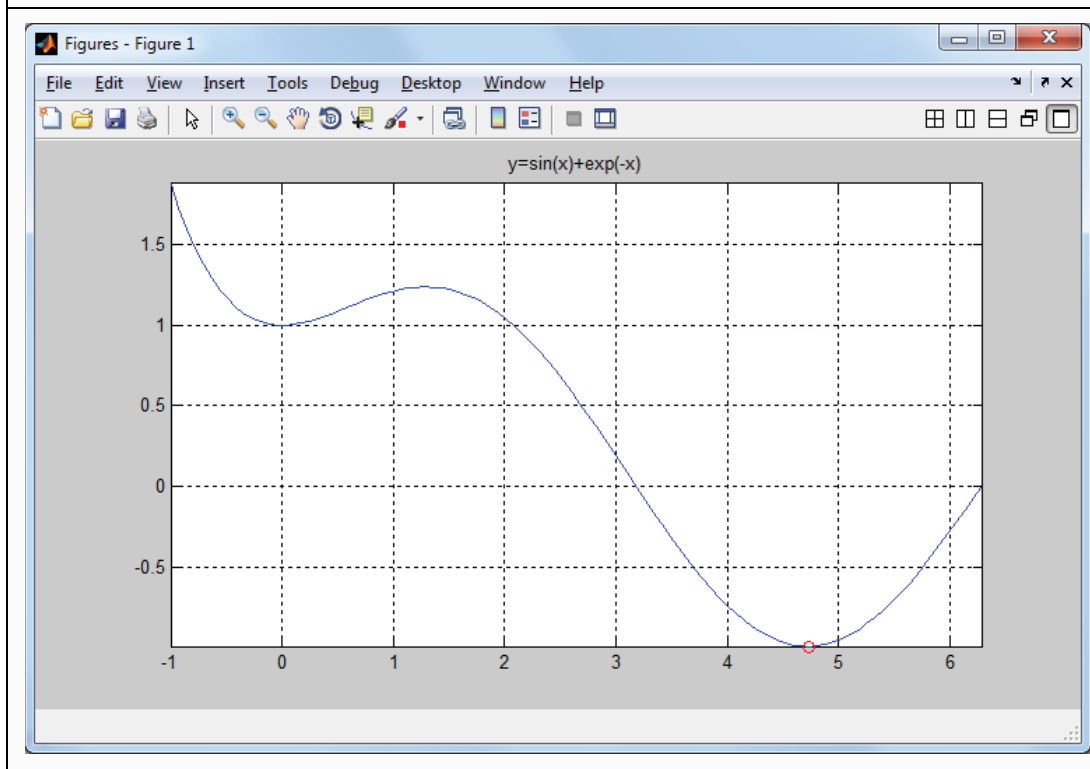
Er du student? Les mer her
www.statnett.no/no/Jobb-og-karriere/Studenter



Since the X- and Y-values are paired, the X-value that corresponds to the minimum Y-value must be in the same location in x . We can then put a red o-marker in the correct spot by issuing

```
>> hold on; plot(x(minLoc),y(minLoc),'ro')
```

Figure 7-3: A function plot

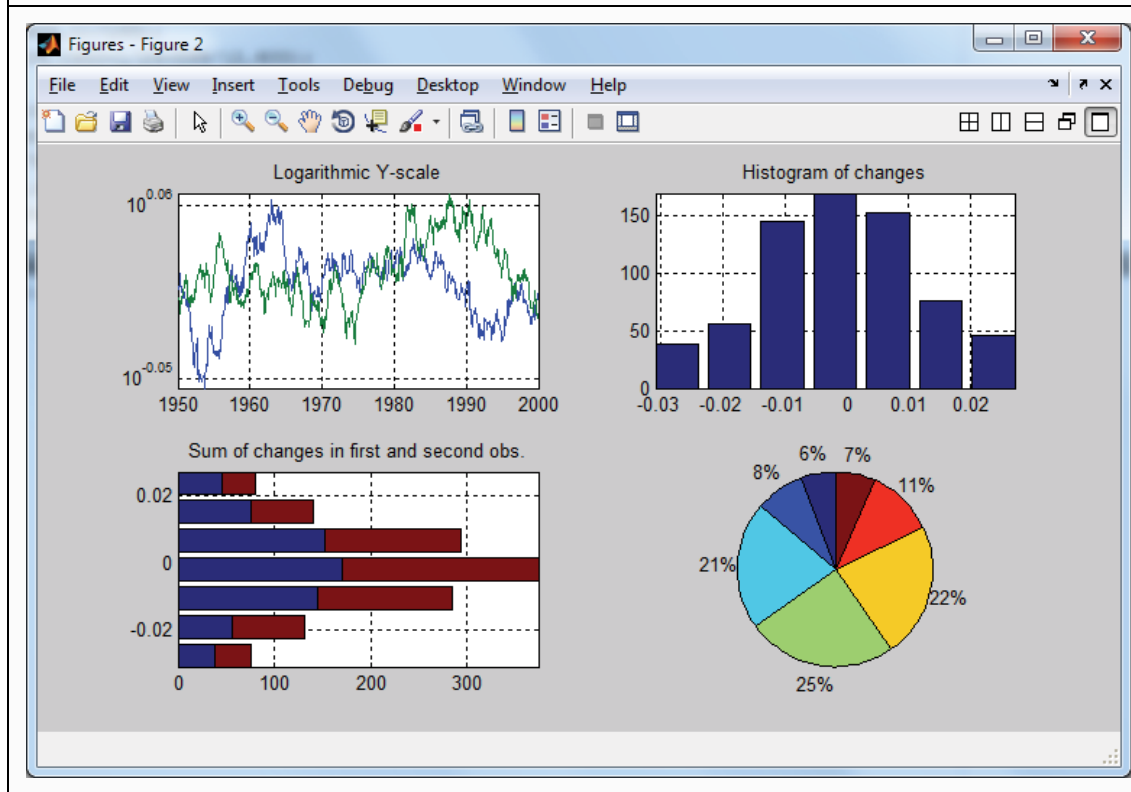


7.4 Several graphs in one window and other types of graphs

It is often useful to plot several graphs in the same figure window. To accomplish this, we use the command `subplot()` that partitions the figure window into several rows and/or columns. At the same time, we introduce a few more graph types. (For descriptions of the graph types, see Section 7.5.) In Figure 7-4 you see the graphs in the partitioned figure window.

```
>> subplot(2,2,1), semilogy(dates,[obs obs_2]), grid
>> title('Logarithmic Y-scale')
>> [nObs bins]=hist(diff(obs),7);
>> subplot(2,2,2), bar(bins,nObs); axis tight, grid
>> title('Histogram of changes')
>> nObs_2=hist(diff(obs_2),bins);
>> subplot(2,2,3), barh(bins,[nObs' nObs_2'],'stacked'),axis tight
>> grid, title('Sum of changes in first and second obs.')
>> subplot(2,2,4), pie(nObs); axis tight
```


Figure 7-4: Subplots of selected graph types



7.5 Other two-dimensional graphs

`semilogx(X)`

Plots X with a logarithmic X-scale.

`semilogy(X)`

Plots X with a logarithmic Y-scale.

`loglog(X)`

Plots X with logarithmic scales on both axes.

`hist(X)`

Histogram

`[nObs bins]=hist(X,7)` partitions the elements of X in 7 groups that are equally spaced, and then returns the number of observations in each group and the mid value of each group. It does not plot anything.

`hist(X,7)`, with no output arguments, plots the histogram. `nObs=hist(X,bins)` uses the midpoints in the vector bins and calculates the number of observations in each group.

`bar(bins,nObs)`

Bar diagram

Draws a bar diagram with the X-values in bins and the height of the bars in nObs. vb

```
barh(bins,nobs)
```

Horizontal bar diagram

Draws a horizontal bar diagram with the X-values in `bins` and the bar widths in `nObs`. If `nobs` is a matrix, it draws as many groups as there are rows.

`barh(bins,nObs,'stacked')` stacks the number of observations instead of grouping them.

```
pie(X)
```

Pie plot

Draws a pie plot. Each slice has the same proportion of the pie as the corresponding value in `X` has of `sum(X)`.

7.6 Plotting tools

You may have thought about whether it is possible to change things like line width or text fonts in graphs. And, yes, this is possible, and you can do it in several different ways. The simplest way of changing graphs that you have already produced, is to use the Matlab plotting tools. This is a graphical interface that you can open with the command `plottools` from the Command Window. Figure 7-5 shows the plotting tools opened with the function `graph` from Section 7.3.



Hva får egentlig en ingeniør- eller teknologistudent for 300 kroner?

- Medlemskap i en aktiv studentorganisasjon – hele studietiden
- 150 tillitsvalgte studenter som ivaretar dine interesser
- Jobbsøkerkurs
- Gratis PC-forsikring og gode bank- og forsikringstilbud
- Teknisk Ukeblad og NITO Refleks
- Møteplasser på web 2.0

Flere medlemsfordeler og innmelding: www.nito.no/student

Alle som studerer på ingeniør-, bioingeniør-, sivilingeniør eller andre teknologistudier (høgskolekandidat, bachelor eller master) kan bli medlem i NITO.

NITO NORGES STØRSTE ORGANISASJON FOR INGENIØRER OG TEKNOLOGER

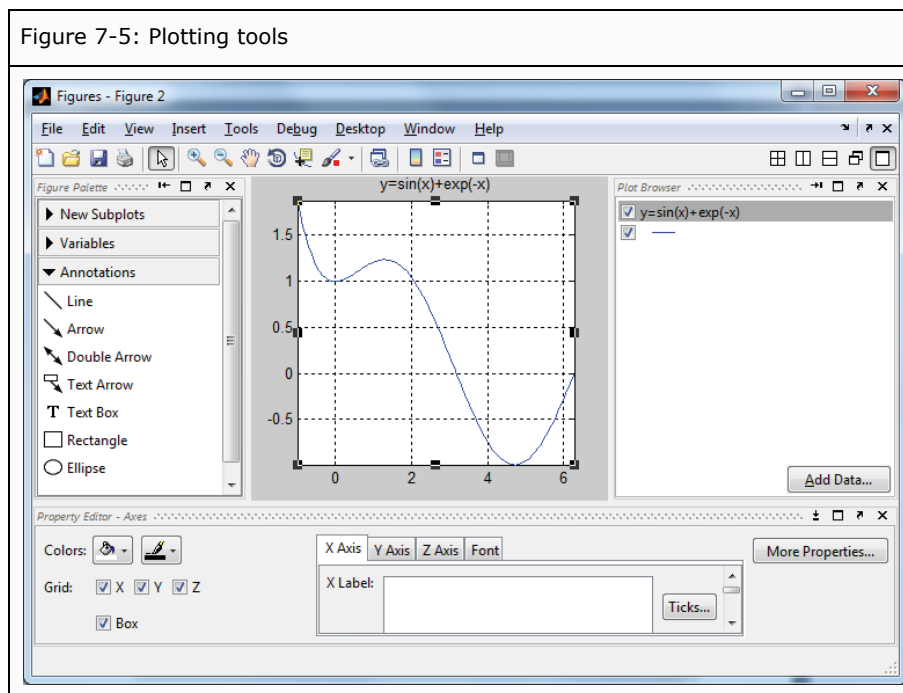


There are three different parts to the interface, the Figure Palette, the Plot Browser, and the Property Editor. If you do not see, for instance, the Figure Palette, select `Desktop > Figure Palette` from the drop-down menus. There are enormous amounts of things you can change with the plotting tools. For instance, in the Figure Palette you can choose lines, figures, or text to add to the figure. In the Plot Browser, you can delete objects, such as the plotted line, by unchecking them in the list.

The Property Editor holds the largest number of tools and the ones shown changes as you select different objects in the graph. From here, you can change labels and how they look (fonts, weight, color, etc.) as well as how the objects in the graph look (line width, line type, color, markers, and plot type). Clicking "More Properties..." in the upper-right corner of the Property Editor opens up a new interface where you can change almost any detail of a graph. Note that, which items are listed in the interface depends on which object you have selected in the graph.

It is a very good idea to spend some time experimenting with the plotting tools. This will give you a good idea of the possibilities, and make it easy to implement certain looks that you want for your graphs in the future.

Note that you can also edit graphs by choosing `Tools > Edit Plot` from the drop-down menus at the top of the window. After having done this, you can click-and-drag object in the graph and select and right-click objects to change their properties. To, for example, change the solid line to a dashed line, you right-click the line and select `Line Style > dash`. Double-clicking an object opens up the Property Editor from the plotting tools.



7.7 More about graphics

- There are several more specialized graph types in Matlab. Try, for example, `compass`, `feather`, `fill`, `polar`, `quiver`, `rose`, `stairs`, or `stem`.
- For a list of commands that relate to ordinary two-dimensional graphics, try `help graph2d`.
- Matlab also supports three-dimensional graphics. For a list of commands, try `help graph3d`.
- For specialized graphs, both two- and three-dimensional, try `help specgraph`.



Skatteetaten



Vil du jobbe i et av landets største IT-miljøer?
Vi skal gjøre det kompliserte enkelt

Skatteetaten tilbyr store fagmiljø og utfordrende oppgaver innen:

- > Systemutvikling
- > Service oriented architecture (SOA)
- > Business intelligence (BI)
- > Testledelse
- > Webutvikling
- > IT sikkerhet
- > Infrastruktur
- > Brukergrensesnitt

For nyutdannede IT-spesialister kan vi tilby et to-årig traineeprogram.

For mer informasjon se skatteetaten.no/jobb

Profesjonell • Nytenkende • Imøtekommende



Programming in Matlab

8 Scripts

So far, we have not been using any automated processes. What we have done is to issue one or a few commands from the Command Window to perform tasks step-by-step. This is one way to use Matlab. If you have a small-scale problem, it can often be solved by issuing just a few commands. Furthermore, if you are unsure of how to go about solving a certain problem, trying a few different approaches, by issuing commands from the Command Window, is often a good first step.

However, there are many situations in which writing a program simplifies the task. Many problems involve repeating the same operations many times. This could be so in different ways. It could be that you want to make new estimates after new data has become available. Then you solve the same problem again, including the new data points. Alternatively, it could be that the problem requires that you perform a simulation, where you repeat the same process many times with random input data. Or it could be that you have a large data set and want to perform the same calculations, again and again, on each data point. In cases like these, writing a program is a good idea.

Matlab differentiates between two types of programs: *scripts* and *user defined functions*. Scripts are similar to automatically executing the same type of command lines as you, up until now, have been issuing manually. All variables defined within the script appear in the Workspace and you can use them after the script has finished.

User defined functions, on the other hand, are similar to the functions you have used, for example `zeros(1,3)` and `sortrows(testMatrix,2)` that we used in sections 4.1.1 and 4.4, respectively. When a user defined function is executed, the operations of the program are hidden from the user, and variables used within the program do not show up in the Workspace and are not possible to use afterwards. You might not even have thought about the possibility that, for example, the function `sortrows()` uses any internal variables. (It does.) The distinction between scripts and user defined functions will probably become clearer as we describe the latter in Section 9.

8.1 The Editor

A program is simply a list of commands. When the program is run, the commands are executed in order from beginning to end. Since a program is just a list of commands, you can use any text editor to write a program, for instance Notepad. However, it is much more convenient to write the program in the Matlab Editor. The Editor is integrated in Matlab and has many features that make programming easier, such as color-coding of selected keywords, automatic row numbers, and a powerful debugger that provides tips on how to correct or improve programs.

You open the Editor by entering the command `edit` from the Command Window, or by choosing `Desktop > Editor`. If you use the second option, you also have to open a new script document in a second step, for instance by choosing `File > New > Script`. If you want to dock/undock the Editor, you click the arrows in the upper-right corner, as described in Section 2.

As mentioned, there is a debugger feature in the Editor. This debugger will highlight parts of the text you write, underline some parts, and, if you hover with the mouse over the underlined parts, give balloon-style tooltips. This is sometimes very helpful, but it can also be quite annoying. In Section 12.1, we will describe how to use this feature. For now, if you want to turn it off, choose `File > Preferences...`, then first click "Code Analyzer" in the menu to the left in the new window, and then uncheck the tick box labeled "Enable integrated warning and error messages". Click "OK" to close the window and return to the Editor.


8.2 Writing a script

Let us use the commands from Section 7.2 that produce a plot of a random series, as an example of how to write a simple script. In the first version of the script, we just want it to reproduce what we did before. Let us collect the lines from before and type them into the Editor.

```
obs = cumprod(1+randn(600,1)/100);
dates = 1950+1/24: 1/12: 2000;
plot(dates, obs, 'g')
title('Levels during 1950 to 2000')
xlabel('Year'), ylabel('Level')
grid
legend('First observations')
```

If you entered the command lines earlier, it is possible to use the Command History window as a shortcut to typing the whole script from the keyboard. Just select the appropriate lines in the window by clicking on them. More than one line can be selected by holding down `<Ctrl>` while clicking new lines. When you have selected the lines you want, you can drag-and-drop them in the Editor or you can copy them and then paste them in the Editor. Alternatively, you can right click one of the selected (and highlighted) lines in the Command History window, and then select `Create Script` from the context menu. In the latter case, Matlab opens a new window within the Editor and copies the lines to that window.

Before we can run the script, we need to save it. It is often convenient to save programs to the Current Folder. Select `File > Save` and the Current Folder is the preselected destination folder. In the dialogue window that appears, choose a name for the script, for example `randomPlot.m`. The script needs the extension `.m` to work, but if you save it from the Editor, the extension is automatically added to the name. If you try to run the program before it is saved, you will be asked to save it first.

After you have saved it, it is possible to run the script. There are a few different ways to do that, and you can choose the one that is most convenient for you. You can click the icon  at the top of the Editor window, you can press F5 while in the Editor Window, or you can issue the name of the script as a command in the Command Window.

```
>> randomPlot
```

In either case, a random plot with years from 1950 to 2000 on the X-axis, axis-labels, and a title appears, just as it did in Section 7.2. If you run the script repeatedly, new plots appear with new sets of random data. As is typical for a script, the variables defined and used within the script, `dates` and `obs`, appear in the Workspace, and you can access them after the script is run. It is also possible to use variables defined outside the script from within the script, if you would like to do that. In short, running a script is just like issuing the commands manually, one-by-one, only faster.

Suppose we want to define the title of the graph manually, outside the script. To do this, we can change the line `title('Levels during 1950 to 2000')` to `title(titleVar)`. After having changed the line, you *must save the script again* to implement the change. After having done this, we go to the Command Window and define the title variable. Then, issue the script name to rerun the program.

```
>> titleVar = 'Level evolution';
```



OLJE- OG ENERGIDEPARTEMENTET



Er du full av energi?

Olje- og energidepartementets hovedoppgave er å tilrettelegge for en samordnet og helhetlig energipolitikk. Vårt overordnede mål er å sikre høy verdiskapning gjennom effektiv og miljøvennlig forvaltning av energiresursene.

Vi vet at den viktigste kilden til læring etter studiene er arbeidssituasjonen. Hos oss får du:

- Innsikt i olje- og energisektoren og dens økende betydning for norsk økonomi
- Utforme fremtidens energipolitikk
- Se det politiske systemet fra innsiden
- Høy kompetanse på et saksfelt, men også et unikt overblikk over den generelle samfunnsutviklingen
- Raskt ansvar for store og utfordrende oppgaver
- Mulighet til å arbeide med internasjonale spørsmål i en næring der Norge er en betydelig aktør

Vi rekrutterer sivil- og samfunnsøkonomer, jurister og samfunnsvitere fra universiteter og høyskoler.

www.regjeringen.no/oed



Se ledige stillinger her

www.jobb.dep.no/oed



```
>> randomPlot
```

The plot now has the new title.

It is very useful to add comments to programs. In the present case, we should at least add a comment with the name of the script at the top, and a brief description of what the script does.

```
% RANDOMPLOT
% This script creates a random series of changes and turns them
% into an index-like series by multiplying cumulatively.
% A series of dates of the same length is also created.
% Finally, the series is plotted against the dates.
```

As you see, comments are, by default, color-coded green.

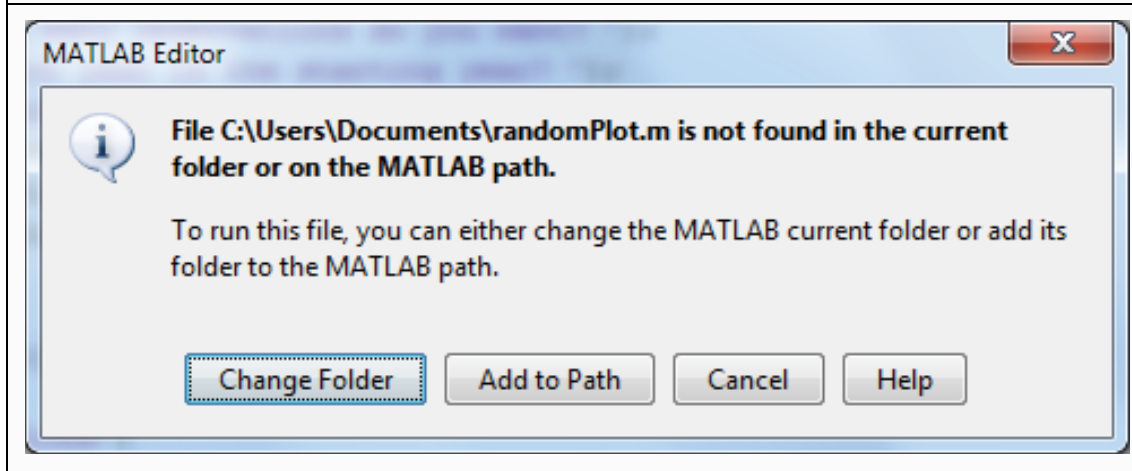
8.3 The search path

Often, it is convenient to save programs to the Current Folder. This is at least true while developing a new program and if you do not have many programs saved there already. However, with time it will become necessary to sort programs in different subfolders. You may also want to save programs in a completely different location in the folder hierarchy. If you just move a program to another location, Matlab will usually not be able to find it, and you will get an error message when you try to run it.

What happens internally when you issue a command is that Matlab starts looking for a match for the command that you have issued. (Compare with the discussion in Section 3.7.) First, it searches the Workspace, and then it searches the Current Folder. If no match is found, it starts searching other folders. However, it does not search the whole computer; it only searches folders that have been added to its “search path”, in the order that they are listed. Therefore, if you want to use a folder that is not already on the search path, you have to explicitly add it.

To open the dialogue for this, choose `File > Set Path...`, alternatively issue the command `pathtool`. To add a new folder to the path, click “Add folder...” and find the folder in the hierarchy that appears in a new window. Select it and click “OK” to add it. If you want the addition to be permanent, so that Matlab will remember the path the next time you open it, you also have to click “Save” before clicking “Close” to exit the dialogue. If you do not save the information, the added folder will only be accessible until the next time you close Matlab.

Figure 8-1: Change folder/add path dialogue



If you try to run a script that is not saved on the path or in the Current Folder, using either F5 or by clicking the run-button, a dialogue is opened. It will ask whether you want to change the Current Folder to the location of the script file, or if you want to add the folder that contains the file to the path (see Figure 8-1). Note that, this does not happen if you issue the script name from the Command Window, as Matlab cannot find the file in that case.

8.4 User interaction with the script

As the program is written, you now have to enter the graph title from the Command Window and you cannot change any other features without changing the script itself. However, suppose you want to be able to produce graphs that are slightly more dynamic. You may want to be able to produce graphs of different lengths as well as with different starting years. It is then convenient if the script lets you enter this information interactively (i.e., that it stops to ask you for input). We add three lines at the beginning of the script that ask for input.

```
nOfObs = input('How many observations do you want? ');  
startYear = input('Which year is the starting year? ');  
titleVar = input('Which title do you want for the graph? ');
```

When we rerun the script, it now first stops to ask for the number of observations and waits until we have entered a number and pressed <Enter>. When we do that, the variable `nOfObs` is assigned the value that we enter. Then, similarly, the script asks for the starting year and a title. Note that you have to enter the title as a string (i.e., enclose it within single quotes). Now, for the program to actually produce what we have asked it to do, we have to change how the variables `obs` and `dates` are created.

The number of observations is easily changed by changing the line `obs = cumprod(1 + randn(600,1)/100);` to `obs = cumprod(1 + randn(nOfObs,1)/100);`. However, for the dates variable, we also have to calculate the value with which to end the matrix. Earlier, we did this manually. The starting value is easy. It will be `startYear + 1/24`. The increment, the value we increase each consecutive observation with, is `1/12`, as before. The last observation must then be `startYear + 1/24 + 1/12*(nOfObs-1)`.

As a new feature, let us also add explicit output from the program, where it specifies the starting year, the ending year, and the number of observations. One way to do that is to delete the semicolons at the end of the lines where the corresponding variables are defined. That, however, produces very ugly output. It is more appealing to use the display command, `disp()`. One problem here, though, is that we need to mix character data with numerical data. To circumvent that, the numerical data has to be translated to character data using the `num2str()` function. Since the information we want to display in each case is a concatenation of two matrices, first a character string and then a translated numerical variable, we have to enclose the two parts within square brackets.

The updated script will then look like this, where the creation of dates is done in two steps.

```
% RANDOMPLOT
% This script creates a random series of changes and turns them
% into an index-like series by multiplying cumulatively.
% A series of dates of the same length is also created.
% Finally, the series is plotted against the dates.
nOfObs    = input('How many observations do you want? ');
startYear = input('Which year is the starting year? ');
titleVar  = input('Which title do you want for the graph? ');
obs = cumprod(1+randn(nOfObs,1)/100);
endYear = startYear+1/24+1/12*(nOfObs-1);
dates   = startYear+1/24: 1/12: endYear;
plot(dates, obs, 'g')
title(titleVar)
xlabel('Year'), ylabel('Level')
grid
legend('First observations')
disp(['Start year: ' num2str(startYear)])
disp(['End year: ' num2str(round(endYear))])
disp(['Number of observations: ' num2str(nOfObs)])
```

Save the script and run it a few times with different input values to see that it works as expected.

9 User defined functions

We have already used functions many times. For example, we used `cumsum()` and `randn()` in the previous section, and we have listed many useful functions throughout the book.

User defined functions are similar to these functions. One similarity is that you call them in the same way. You type the function name and enclose one or several arguments within parentheses after the name, for example `randn(5,2)`, which uses two input arguments and calls a function that creates a 5x2 matrix of random numbers. Another similarity is that variables used within the function, do not show up in the Workspace after the function has been executed. You could think of this as that the inner workings of the function are hidden from the user. This is often an advantage, as it prevents the Workspace to become cluttered with variables that you do not need.

Unlike scripts, user defined functions must begin with a declaration that it is a function, and you must specify input and output variables. The general form of this declaration is

```
function [output] = functionName(inputArguments)
```



HELT GRATIS!

S for Skikk & Bank

DU FÅR BOKA
HOS DNB

En bok om ting som er greit å vite når du har flyttet hjemmefra.

dnb.no

DNB

Bank fra A til Å



Here, the word `function` states that the program is a user defined function, `output` is one or a list of several variables that the function returns, `functionName` is a name that the user gives to the function and that is later used to call it. `inputArguments` is one or a list of several variables that the function uses as input. If there are several input arguments they must be separated by commas, but output variables can be separated by either commas or blank space. If there are more than one output variable, they must be enclosed within square brackets, but if there is only one, no brackets are needed. (Note that all these requirements adhere to the syntax of functions described in Section 3.3.) The word “function” is color-coded blue, to show that it is a keyword in Matlab. The help text at the beginning of the function also has special uses, which are described later in sections 13.1.1 and 13.1.2.

Let us write a simple function to show how this works. In the previous section, we created a matrix of the cumulative product of random numbers, where the desired number of observations was supplied by the user (i.e., the line `obs = cumsum(randn(nOfObs,1))`). This could be done with a simple function. The input argument is just one number, the desired number of observations, and the output is a matrix, similar to the one from the previous section. So, to write the function, open a new window in the Editor, enter the following code, and then save it.¹³

```
function obs = randomIndex(nOfObs);  
% RANDOMINDEX  
% Produces an index with random changes and nOfObs elements.  
obs = cumprod(1 + randn(nOfObs,1)/100);
```

When saving a new function, Matlab always suggest that you save it under the name you have entered as function name. This is good practice. If you use different names, then the function can only be called using the name you saved it with, so the name in the text of the function will be meaningless.

Now, in the Command Window, issue the function as a command. (A function cannot be executed using F5 or clicking the run-button.)

```
>> randomIndex(5)  
ans =  
    -1.3489  
    -0.045377  
    -1.665  
    -1.9938  
    -3.6051
```

There are a few things to note here. As already mentioned, the variables within the function are local (i.e., they only exist within the function). When you call the function using `randomIndex(5)`, the value 5 is supplied to the function, and there the variable `nOfObs` is assigned the value 5. Then, inside the function, a new variable, `obs`, is defined and it is set equal to the cumulative product of a set of random variables. Lastly, the value of `obs` is returned from the function to the Workspace. Since we have not entered a variable name, Matlab sets the variable `ans` equal to the result. None of the variables `nOfObs` and `obs` is accessible outside the function. To make this obvious, we clear the Workspace from all variables and reissue the command.

```
>> clear
>> numbers = randomIndex(5);
```

The only variable listed in the Workspace now is the new variable `numbers`. Note, also, that variables in the Workspace are not accessible from within the function either. Therefore, all information that the function needs must be entered in the form of input arguments, or be imported into the function by some other method, for instance from an Excel file as in Section 6.6.2.

Note that, if the function has any output, the output variable(s) must be explicitly defined within the function. Not all functions have input or output, though. For instance, a function might use random data that is defined within the function, and the output could be a graph instead of variables.

We can now change the script from Section 8.4 to use our new user defined function. Just change the line `obs = cumsum(randn(nOfObs,1));` to `obs = randomIndex(nOfObs);`.

9.1 About the differences between scripts and user defined functions

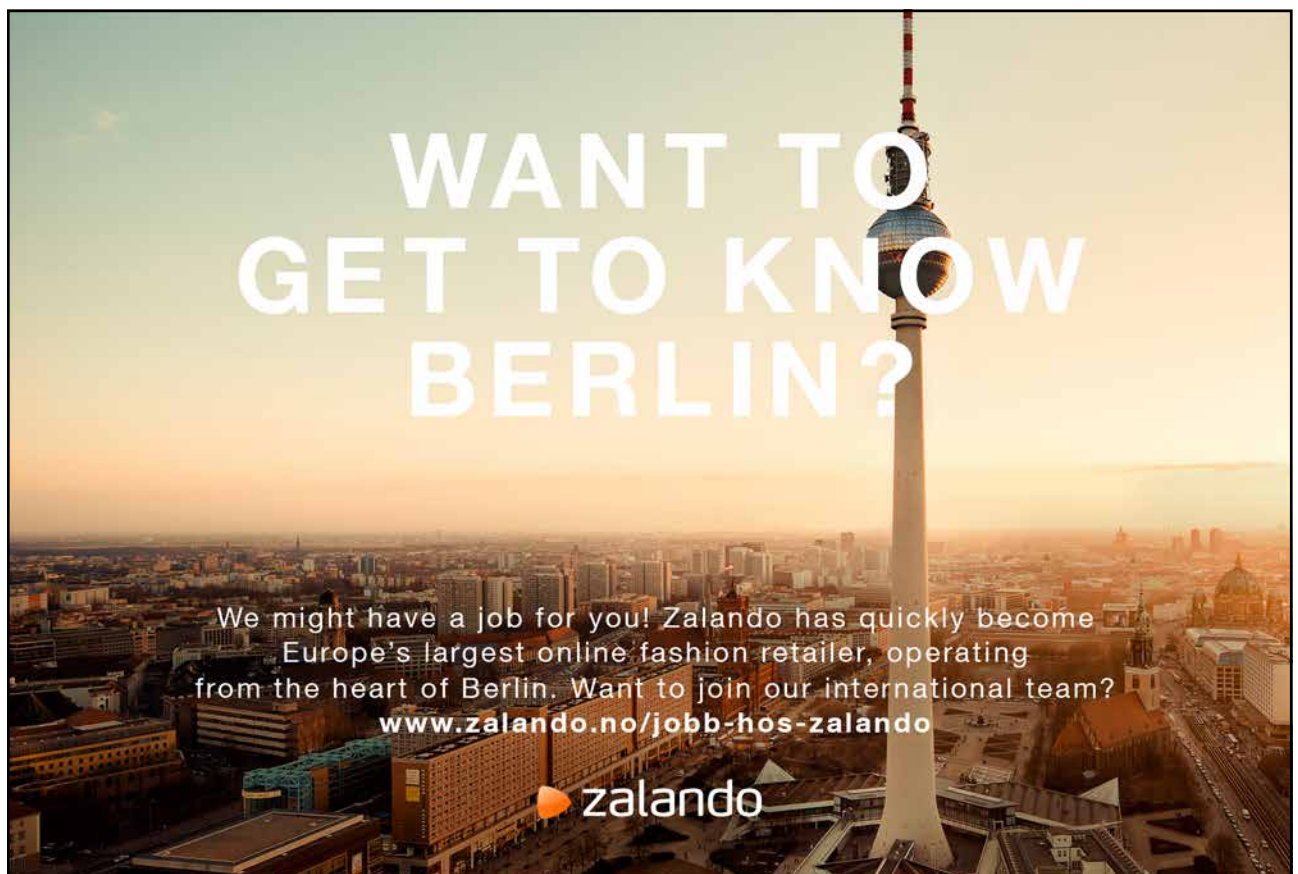
When should you write a script and when should you write a function? Consider first the main differences between the two.

- Scripts use the existing Workspace whereas functions use their own. Variables used in functions are consequently hidden from the user and are not accessible outside the function.
- Functions usually return explicit output whereas scripts do not.
- Running scripts is similar to rerunning code you have issued from the Command Window. Running a user defined function is similar to executing Matlab functions.

These differences make for the following rule of thumb, which usually, but not always, is correct: If you are solving a general problem, use a function; if you are solving a specific problem, use a script.

9.2 More about functions

- You can open functions directly into the Editor by issuing `edit functionName`. You can also open many Matlab functions this way. Note, however, that many of the basic functions are built-in functions and these are not viewable. Try, for instance, `edit mean` to see how Matlab calculates the mean.
- Matlab also has a concept called “anonymous functions”. This is a way to enter a function quickly without storing it to a file. See `help function_handle` and search “anonymous functions” in the Help Browser. (The Help Browser is described in Section 13.2.1.)



10 Flow control

The programs we developed in sections 8 and 9 were executed in a linear fashion (i.e., execution started with the first line of code and then moved to the second line, and so forth, until the last line was reached, after which it ended). Oftentimes we want to make programs that are more complex than that, though. First, we want to be able to repeat specific tasks many times, possibly with minor changes each time. That is convenient to do in a so-called loop. Second, we want the program to be able to make decisions, so that, depending on different circumstances, it performs different operations.

Before we begin, it is good to learn the key combinations <Ctrl>-C or <Ctrl>-<Break>. Both of these break the execution of a program. Sometimes it happens that, by a programming mistake, a program goes into an infinite loop, or it might go into a loop that will take more time to execute than you are willing to wait. Then it is essential to know how to get out of that.

10.1 Loops

A loop is a collection of commands that is repeated a certain number of times. There are a few different ways to do this. One of the most frequently used is the for-loop. This is particularly useful when you want to repeat a task a certain number of times, say exactly 10 times.

The simplest form this command takes is

```
for counter = startValue:increment:endValue
    :
    :
end
```

Here, `for` indicates that a for-loop begins and `counter` is the name of the so-called counter variable. The part `startValue:increment:endValue` is a matrix created using the colon operator. Finally, `end` indicates where the loop ends. The part between the `for` and the `end` statements is what is repeated. Note that, the `for` and the `end` statements, when written by themselves on separate rows, do not have to be succeeded by a comma or a semicolon as most other commands do. Note also that, for-loops can be nested, so that there can be a loop within another loop.

To give an example of a simple loop, consider the case of the Fibonacci numbers. This is a series of numbers where the first two are equal to one, and then each consecutive number is equal to the sum of the previous two. The first twelve are then 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, and 144. Say we want to write a user defined function that produces a vector of Fibonacci numbers, where we can choose how many elements we want. This can be easily done in a for-loop.

```

function fib = fibonacci(fib_max)
% FIBONACCI
% Produces a column vector of fib_max Fibonacci numbers.
fib = ones(fib_max,1);
for fib_nr=3:fib_max
    fib(fib_nr,:) = fib(fib_nr-1,:) + fib(fib_nr-2,:);
end

```

This code first defines a column vector of ones to hold the Fibonacci numbers. Then, starting from the third number, it calculates the sum of the preceding two numbers, and then repeats this process until the last requested number is reached. Then it returns the vector of Fibonacci numbers.

Note the line `fib = ones(fib_max,1);`. What this does, besides setting the first two numbers equal to 1, is to preallocate space in a vector that will hold all the numbers. This ensures that the vector is large enough to hold all numbers already from the beginning. Suppose line was changed to `fib(1,:) = 1;` `fib(2,:) = 1;`. The program would then produce exactly the same output vector. However, at each new step, the vector would grow by one element, forcing Matlab to redefine it every time. And this takes much more time. Therefore, preallocating in this type of situation is a standard procedure in Matlab.

As you see, the line within the loop is indented. This useful convention makes the code easier to read. However, it does not change the way the code is executed. If you write the code in one single line (in which case you have to end the `for`- and `end`-statements with a comma or a semicolon), the output is the same. It will be more difficult to read, however. When you type the code into the Matlab Editor, the lines after a `for`-statement will automatically be indented.

Usually, loops appear within programs. However, it is quite possible to use them in the Command Window as well. If you want to type it there, using several lines before starting execution, instead of ending the lines with `<Enter>`, you press `<Shift>-<Enter>`. This just moves the cursor to the next line without starting execution. At the last line, you press `<Enter>` and Matlab executes the whole code.

As an alternative, you can end each line with `<Enter>`. Then, when issuing the first three lines of code, nothing seems to happen, except that the `>>` prompt, that usually appears at the beginning of each row, disappears. This is because Matlab waits for the `end` command that concludes the loop. Before that has been issued, Matlab does not know what to repeat. When, finally, the concluding `end` is issued, the whole loop is executed.

Yet another alternative is to enter the whole set of commands as one long line in the Command Window, separated by commas or semicolons. To give an example and, at the same time, give an idea of how time consuming it is not to preallocate space for variables that grow inside loops, we issue the following code in the Command Window.

```
>> tic, for run_nr=1:10000; fibonacci(1000); end, toc  
Elapsed time is 3.873173 seconds.
```

Here, `tic` saves the time when it was issued and `toc` calculates the amount of time that has passed since `tic` was issued, thereby imitating a stopwatch. Apart from that, a vector of the first 1000 Fibonacci numbers is created, and this is repeated 10000 times. All this takes approximately 3.9 seconds. Then we change the function line where `fib` is preallocated to `fib(1,:) = 1; fib(2,:) = 1;`, save the function, and rerun the code to time it again.

```
>> tic, for run_nr=1:10000; fibonacci(1000); end, toc  
Elapsed time is 17.703510 seconds.
```

Evidently, we save about 14 seconds, or almost 80%, in this case.

10.2 Relational and logical operators

Conditional statements are statements that are executed only if certain conditions are fulfilled. Hence, they enable programs to make decisions. Operators that are used in conditional statements fall in one of two categories, relational operators and logical operators. We have already used some of these in sections 3.5.2 and 4.2.3. Remember that the outcomes of operations with relational or logical operators are *true* or *false*, and that these are indicated by 1 or 0.



"I studied English for 16 years but...
...I finally learned to speak it in just six lessons"

Jane, Chinese architect

ENGLISH OUT THERE

Click to hear me talking before and after my unique course download



Relational operators

<code><</code> , <code><=</code>	“Is less than” and “is less than or equal”, respectively. For example, <code>2 < 3</code> is true and produces the output 1.
<code>></code> , <code>>=</code>	“Is greater than” and “is greater than or equal”, respectively. <code>2 > 3</code> is false and produces the output 0.
<code>==</code>	Is equal to. (Note: There are <i>two</i> equality signs to separate this operator from the assignment operator). <code>5 == 5</code> is true.
<code>~=</code>	Is not equal to. <code>5 ~= 5</code> is false.

Logical operators

<code>&</code>	Logical and This is true if both the statement to left and the one to the right of <code>&</code> are true, else it is false. <code>2<3 & 5>4</code> is true, since both <code>2<3</code> and <code>5>4</code> are true. When comparing scalars, Matlab prefers double <code>&:s</code> , as this has additional functionality. In the statement <code>0 & 4</code> , for example, the first part is false (0 is false). Then the rest of the statement does not have to be evaluated at all, since an and-statement where one element is false, by necessity has to be false. If you change this to <code>0 && 4</code> , Matlab takes advantage of this fact and only checks the second part when it is necessary (i.e., when the first part is true). Another benefit of this is that, if the second part is a variable that may be undefined, it does not have to be checked if the first part is false, thereby preventing an error message and termination of the program.
<code> </code>	Logical or This is true if either the statement to left or the one to the right of <code> </code> is true or if both statements are true, else it is false (i.e., it is only false if both statements are false). The character <code> </code> is usually produced by typing <code><Ctrl>-<Alt>-<</code> (i.e., the button for “less than”). <code>2<3 5>4</code> is true, since, again, both <code>2<3</code> and <code>5>4</code> are true. Similarly to <code>&</code> , when comparing scalars, it is preferable to use double <code> :s</code> . Then the second part is only evaluated if the first part is false.
<code>~</code>	Logical not This is true if the statement to the right of <code>~</code> is false, and false if it is true. The character <code>~</code> is usually produced by typing <code><Ctrl>-<Alt>-`</code> (i.e., the button for “dieresis”, often located to the left of the button <code><Enter></code>). For example, <code>~ (3<2)</code> is true, since <code>3<2</code> is false.

Note that all numbers, except zero, are true. Zero is false. For example, the statement $2 \ \& \ 3$ is true while the statement $2 \ \& \ 0$ is false.

Also note that, matrices are compared element-by-element, as we saw in Section 5.1. So the statement $[1 \ 2] \ > \ [0 \ 5]$ evaluates to a logical 1x2 matrix, where the first element is true and the second is false: $[1 \ 0]$.

We can now extend the list of the order of precedence from Section 3.1 to include the relational and logical operators.

1. parentheses
2. exponentiation
3. logical not; \sim
4. multiplication, division
5. addition, subtraction
6. relational operators
7. logical and; $\&$
8. logical or; $|$

As before, if there is a tie, the expression is evaluated from left to right. Again, if you want to be absolutely certain about the order, use parentheses, which can also improve readability.

For example, the statement $\sim (0 \geq 0)$ is false. This is because the expression within the parentheses, $0 \geq 0$ (which is true), has precedence over \sim (which makes the true statement false). However, the statement $\sim 0 \geq 0$ (without parentheses) is true. This is because \sim now has precedence over \geq . ~ 0 is first evaluated to be 1, and $1 \geq 0$ is true. In this latter case, note that when the operator needs numerical data (\geq compares numbers, not true/false), the logical variables are interpreted as numbers (i.e., true is interpreted as 1 and false as 0).

10.3 Conditional statements


We can now use the relational and logical operators to write conditional statements. The form of a conditional statement is

```

if ...
    :
    :
elseif ...
    :
    :
else
    :
    :
end

```

The command `if` indicates that a conditional statements begins. After this, a condition that is true or false follows on the same line. On the following lines(s), there is code that is executed if the condition is true. After this follows either an `end`-statement or an `else`-statement. If there is an `else`-statement, the code following the statement is executed if and only if the initial condition is false. The `else`-statement comes in two versions. Either it is an unqualified `else`-statement, or it contains additional conditions. In the latter case, it is issued as `elseif` and followed by the additional conditions. The whole section must end with an `end`-statement, so that Matlab knows where the conditional part ends.



WHILE YOU WERE SLEEPING...

www.fuqua.duke.edu/whileyouweresleeping

DUKE
THE FUQUA
SCHOOL
OF BUSINESS



The following function provides an example of using conditional statements. The input is the score from a test, and the output is a grade from A to C, or “fail”, depending on how high the score was.

```
function grade = score2grade(testScore)
% SCORE2GRADE
% Calculates grade from the score at a test.
if testScore >= 90
    grade = 'A';
elseif testScore >= 80
    grade = 'B';
elseif testScore >= 70
    grade = 'C';
else
    grade = 'fail';
end
```

In the fourth line, the value of `testScore` is compared to 90. If it is greater than or equal to this number, it continues with the following line until it reaches the statement `elseif`, where it skips everything until the end-statement. If, on the other hand, `testScore` is not greater than or equal to 90, the program skips all lines until it reaches the first `elseif`-statement, where it compares `testScore` to 80. Again, if it is greater than or equal to 80, it continues with the following lines until it reaches the next `elseif`-statement, where it skips to the end, etc. Note that this implies that grade B is assigned test scores between 80 up to, but not including, 90. If none of the conditions applies, the grade is set to fail.

Note that only the `for`- and `end`-statements are required; the `elseif`- and `else`-statements are optional. Note also that, it is possible to nest conditional statements, so that you can have another `if... else... end` structure within another.

10.4 More about flow control

- Note that the argument in a `for`-loop (i.e., the `1:5` part of `for counter = 1:5`) is a vector. You do not have to use a vector that is created with the colon operator. You can, for example, enter a vector manually, such as `counter = [1 5 2 4 3]` if this is the order you want. You can even enter a matrix as argument, in which case the loop-variable becomes a column vector, and is executed as many times as there are columns in the matrix.
- In Matlab, ordinary loops can often be replaced with vector operations. For instance, creating a column vector of squares from the numbers 1 to 100 can be done in a loop. Nevertheless, it is faster and simpler to do it as `(1:100)'.^2`.
- Another type of loop is the `while`-loop. This kind continues to loop until a certain condition is not fulfilled any more. For instance, `while x<10... end` loops as long as `x` is less than 10. This is useful when the number of loops is indeterminate. See `help while` for more information.

Download free eBooks at bookboon.com

- Note that it is slightly complicated to compare strings in conditional statements. For example, `'YES'=='Yes'` evaluates to `[1 0 0]`, since each letter is compared individually, element-by-element. Furthermore, letters in small caps and letters in large caps are not equal to each other. To compare strings, use a construction like `isequal(upper('Yes'),upper('YES'))`. `upper()` changes text to upper case and `isequal` compares the whole string instead of the individual elements.
- The command `switch` can be used for executing different code parts depending on a small number of different cases. See `help switch`.
- Loops can have certain conditions inserted where they either break the loop completely, or move on to the next loop without finishing the present one. See `help break` and `help continue`. Another type of break is to leave the entire program that is being executed. For that, see `help return`.



Vi vokser i Norge
og har virksomhet
helt frem til 2050

Er du interessert i sommerjobb
eller fast stilling?



Se informasjon om sommerjobber på
www.bp.no

The advertisement features a large portrait of a young man on the left. To his right, there is a white box containing the text 'Vi vokser i Norge og har virksomhet helt frem til 2050'. Below this, there is a smaller image of an offshore oil rig with the text 'Er du interessert i sommerjobb eller fast stilling?'. In the bottom right corner, there is the BP logo and the text 'Se informasjon om sommerjobber på www.bp.no'.



11 Numerical analysis and curve fitting

Now that we have learnt to program functions, we are able to use Matlab's numerical analysis and curve fitting tools. Here, we describe how to numerically solve equations and find local minimum points, as well as how to perform numerical integration.

11.1 Solving equations

There are several ways to define functions in Matlab. In Section 9, we learnt how to write user defined functions. Let us use the function we plotted in Figure 7-3, define that as a user defined function, and then solve it (i.e., find the point where the function equals zero). We write the function, named `waveFunction`, such that we supply an X-value and then get back the corresponding Y-value. For such a simple function, this is easily done.

```
function y = waveFunction(x)
y = sin(x) + exp(-x);
```

To start searching for a solution, we need a starting value. Looking at Figure 7-3, we see that the function equals zero where x is roughly equal to 3. Then we issue the command `fzero()` as¹⁴

```
>> x_zero = fzero('waveFunction',3)
x_zero =
    3.1831
```

The first input to `fzero()` is the name of the function we want to solve, within single quotes, and the second is a “best guess” where a solution will be found. The present function has many solutions, but only one that is close to 3. Entering other start values can produce other solutions. The output, `x_zero`, is the value of x close to 3 that makes y equal to zero. In our case, the minimum point is found here x equals 3.1831.

11.2 Finding a function minimum point

It is equally easy to find a local minimum of a function. We see in Figure 7-3 that there should be a local minimum where the red circle marker is, somewhere between 4 and 5. The command for finding a minimum between two values on the X-axis is `fminbnd()`.

```
>> [x_min,fval] = fminbnd('waveFunction',4,5)
x_min =
    4.7213
fval =
   -0.99106
```

The second and third input arguments are the end values of the region along the X-axis where we want to search for a minimum. One is found at 4.7213, and the corresponding Y-value, fval, at that point is -0.99106. (Note that, this is slightly smaller than the value we found when plotting the minimum point in the figure in Section 7.3. The difference depends mainly on how fine grained the x-vector was there.)

11.3 Numerical integration

To numerically integrate our function between -1 and 2π (i.e., the plotted region), there are several different methods to choose from. For instance, we can issue

```
>> area = quad('waveFunction',-1,2*pi)
area =
    2.2567
```

The area beneath the function is approximately 2.2567.

11.4 Curve fitting

Matlab also has a convenient tool for curve fitting. If we have two vectors, x and y , with paired observations, we can approximate the functional relation between them with a polynomial of some degree. If the degree is 1, the relation is linear; if it is 2, the relation is quadratic, etc. This can be done with the function `polyfit()`. The following script estimates the coefficients of polynomials of order 1, 2, and 3, for a given set of observations, and plots the results in three graphs.

```
x = [1 2 3 4 5 6 7 8 9]; y = [2 3 3 5 7 8 8 9 7];
x_val = linspace(0,10,100);
for degree=1:3
    poly = polyfit(x,y,degree);
    disp(['Coeff., case ' num2str(degree) ': ' num2str(poly)])
    y_val = polyval(poly,x_val);
    subplot(3,1,degree)
    plot(x,y,'r*'), axis([0 10 0 10])
    hold on
    plot(x_val,y_val)
    ylabel(['Degree: ' num2str(degree)])
end
```

The output is

```

Degree 1:    0.85      1.5278
Degree 2:   -0.12229   2.0729   -0.71429
Degree 3:   -0.053872  0.68579   -1.3318    2.8413

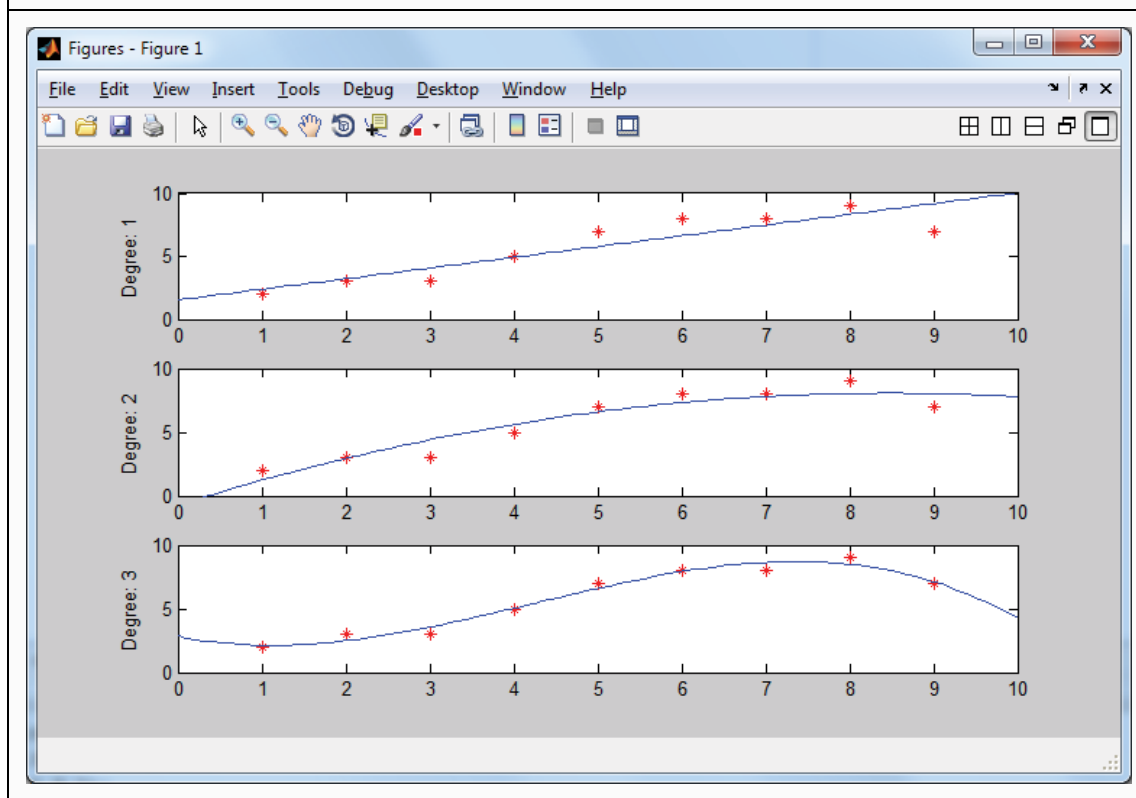
```

The first two inputs to `polyfit()` are the vectors of X- and Y-values, and the third is the degree of the polynomial (i.e., the highest value of the exponent). The function responds with a matrix that holds one more element than the degree. The elements of the matrix are the coefficients of the estimated polynomial. For example, in the third case above, the output should be interpreted as $y = -0.053872x^3 + 0.68579x^2 - 1.3318x + 2.8413$.

Matlab also provides a tool for calculating the values of a polynomial, given a vector of coefficients. The function `polyval()` uses a matrix of coefficients, `poly` above, and returns Y-values for given X-values. As in the above case, the X-values can be a vector. Figure 11-1 shows the resulting three plots. The red markers are the same in all three cases, but the curves correspond to the fitted polynomials.



Figure 11-1: Examples of curve fitting



11.5 More about numerical analysis

- There are several more methods for numerical integration. See, for instance, `help quadgk` and `help quadl`.
- Matlab also supports several methods to solve differential equations. Information is best found in the Matlab documentation. See Section 13.2.1 for tips on how to search it.

Debugging and Help

12 Debugging

New programs rarely work as intended the first time. Unfortunately, finding errors can be very difficult and time consuming, sometimes more so than writing the first version of the program. The process of correcting programming errors is called debugging, and Matlab has several tools to help you to do this.

The most obvious sign that your program needs debugging is that you get one or several error messages when running it. Most often, these messages contain useful information about what has gone wrong and, therefore, hints about what you should do to correct the underlying problem. The most straightforward way to proceed, if you do not immediately know what to do, is to search for help on the function that caused the error message (i.e., type `help` and the function name).

However, the irritating truth is that, even if you do not get any error messages, there may be, and often are, errors in your code. Therefore, debugging has to be done even if you do not get any error messages. What you have to keep an eye on is unexpected behavior in the program. If you do not see what you expected, double check if it is your expectation that was mistaken or if is the program that is not working correctly.

The advertisement for GaiTEYE features a background image of a person running on a path during a sunrise or sunset. The GaiTEYE logo, consisting of a yellow square with a stylized 'G' and the word 'gaiteye' in white, is in the top left. Below it is the tagline 'Challenge the way we run'. In the center, the text 'EXPERIENCE THE POWER OF FULL ENGAGEMENT...' is displayed above a horizontal dotted line. To the left, the text 'RUN FASTER. RUN LONGER.. RUN EASIER...' is shown. On the right, a yellow button contains the text 'READ MORE & PRE-ORDER TODAY' and 'WWW.GAITEYE.COM', with a hand cursor icon pointing at it. A target graphic with a hand cursor is also positioned over the runner's feet.



That being said, many errors are quite simple and straightforward. They can be misspellings, forgotten semicolons, or mistaken syntax that Matlab cannot interpret. These types of errors can be checked by automatic routines; the Code Analyzer in Matlab. Furthermore, many errors can be discovered if you execute the code line-by-line, which is also possible.

12.1 The Code Analyzer

The Code Analyzer is the tool that you possibly turned off in Section 8.1. If you did, turn it on again by choosing `File > Preferences...`, then first click "Code Analyzer" in the menu to the left in the new window, and then check the tick box labeled "Enable integrated warning and error messages". Click "OK" to close the window and return to the Editor.

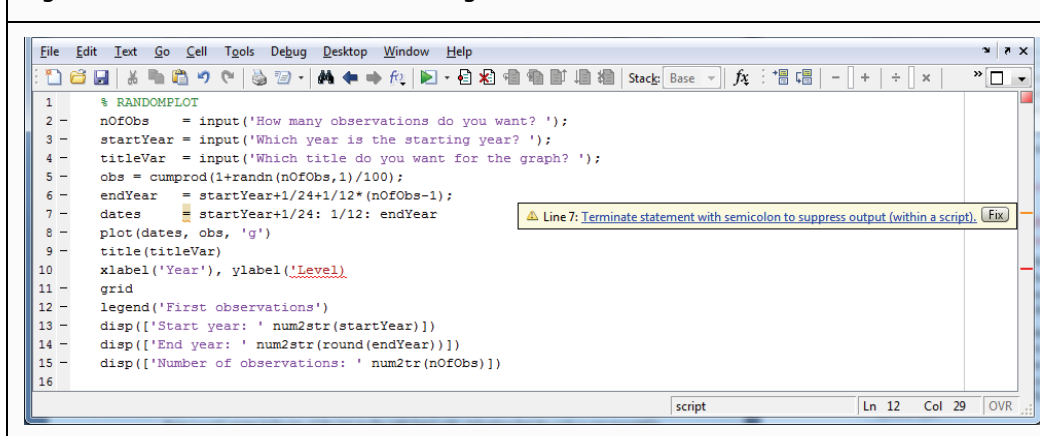
The Code Analyzer checks the code in the Editor against a list of common errors. Where it finds possible errors, the corresponding code is underscored and a small red or orange line is drawn in the list on the right hand side of the Editor. Furthermore, a red/orange square appears at the upper-right corner of the Editor window.

The Code Analyzer distinguishes between warnings and errors, where warnings are indicated in orange and errors in red. A warning indicates that the code is executable, but that the Code Analyzer has identified a piece of code that probably is different from what you intended. An error indicates that Matlab cannot interpret that piece of code and that, if you try to execute it, you will get an error message.

Figure 12-1 shows part of the code from Section 9, but with a few mistakes inserted. You can see that there is a red square at the top of the list on the right hand side, indicating that the code is not executable. Further down on the list, there is first an orange line, indicating a warning on the corresponding line of code, and then a red line, indicating an error. Furthermore, parts of the code on lines 7 and 10 are underlined and the assignment operator in line 7 is highlighted. These indications correspond to the warning and error lines in the list.

If you hover with the mouse over one of the lines in the list, a tip about the error is displayed. Sometimes, as in the figure, there is also an option to fix the problem. Code where this option exists is highlighted. If you click "Fix" in Figure 12-1, the Code Analyzer adds a semicolon after `endYear`. Moreover, if you click the red square at the top of the list, the cursor moves to the next warning/error. Clicking the square in Figure 12-1 will first move the cursor to before the assignment operator on line 7 and then, if you click it again, to before the single quote that precedes the word "Level" on line 10.

Figure 12-1: The Editor with warning and error



In line 10, the Code Analyzer is not able to see where the string is supposed to end, and no automatic fix is suggested. If you add a single quote in the right place, the square in the upper-right corner turns green to indicate that the Code Analyzer accepts all the code.

Note, however, that there is also an error in line 15 where `num2str()` has been misspelled as `num2tr()` (skipping the “s”). If you try to execute the code as it is, you will get an error message, even though the Code Analyzer signals green.

```
??? Undefined function or method 'num2tr' for input arguments
of type 'double'.
Error in ==> randomPlot at 15
disp(['Number of observations: ' num2tr(nOfObs)])
```

Error messages usually have this three-part structure. The first line identifies what Matlab does not understand: it is the part ‘`num2tr`’. The second line identifies where the error occurred: it is in line 15 of the (user defined function) file `randomPlot`. The underlined part is a clickable link to where the error occurred. The last line contains a copy of the whole line where the error occurred (i.e., line 15 in the file).

Sometimes the Code Analyzer identifies a problem where there is none. Suppose, for instance, that you really do not want a semicolon in line 7 in the code. You can then turn off the warning on this line by right-clicking the underscored assignment operator and then choosing `Suppress "Terminate statement with semicolon..." > On This Line`. If you do, a comment (`%#ok<NOPRT>`) is inserted at the end of the line. This instructs the Code Analyzer to ignore warnings of this specific type on this line.




The Code Analyzer is not perfect. It does not find all errors, and it sometimes provides warnings where the code is ok. Nevertheless, it is a very useful tool for debugging code.

Download free eBooks at bookboon.com

12.2 Executing part of the code with F9

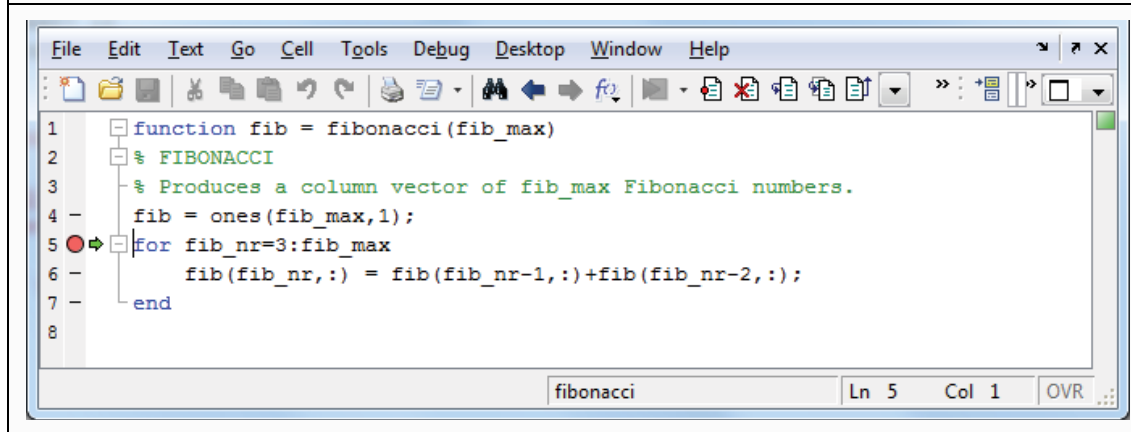
Small pieces of code are much easier to evaluate than large ones. Therefore, it is convenient to be able to execute smaller subsets of the code in the Editor. Select the code you want to execute and press F9 (function key 9). The selected code is copied to the Command Window and executed. This is very useful if the code is not working properly, and you want to see where you get suspicious results. Note that, if you are working with a function (and not a script); the Workspace is not the same when you are executing the code using F9 as when you execute the function in the normal way. You can overcome this problem by defining the variables in the user Workspace as well.

12.3 Using breakpoints

In some cases, there is a lot of code that precedes the part where you think that there is an error. It is then convenient to run the code up to that point and then stop. Setting a breakpoint accomplishes exactly that. Move to the line in the Editor where you want to interrupt the code. Then set a breakpoint on that line by either clicking the short horizontal line between the code and the row number in the left margin, or by clicking  in the toolbar at the top of the Editor. Alternatively, you can press F12 (function key 12) or select `Debug > Set/Clear Breakpoint`. When a breakpoint is set, a red circle appears instead of the horizontal line to the left of the code in the line where the breakpoint is. Note that, the program must have been saved before it is possible to set breakpoints. To delete the breakpoint, you can either click the red circle, click  again, press F12, or select `Debug > Set/Clear Breakpoint`. If you have many breakpoints set and want to delete them all, you can click  or select `Debug > Clear Breakpoints in All Files`.

Let us check the code for the Fibonacci numbers from Section 10.1, using break points. We set a breakpoint in line 5 of the code and execute it by issuing the command `fibonacci(5)`. In Figure 12-2 we see what the Editor looks like. The red circle in line 5 indicates that a breakpoint is set there, and the green arrow next to it indicates that the execution of the code has stopped on that line.

Figure 12-2: Breakpoint



In the Command Window, we see the following.

```

>> fibonacci(5)
5 for fib_nr=3:fib_max
K>>

```

Strømmen produseres ofte langt fra der den skal brukes.

Statnett sitt oppdrag er å gjøre strømmen tilgjengelig, uansett hvor i dette langstrakte landet du bor. Det er vi som bygger og driver "riksveiene" i norsk strømforsyning. Gjennom vårt landsdekkende nett sørger vi for en sikker fordeling av strøm mellom nord, sør, øst og vest.

Vi binder Norge sammen

Statnett
Vårt felles kraftnett

Er du student? Les mer her
www.statnett.no/no/Jobb-og-karriere/Studenter



The first line is the issued command. The second line indicates that the next line to be executed is line 5 and displays the code in that line. The underlined row number is clickable. Note that, on the third line, the Matlab prompt, `>>`, is now preceded by a `K`, so that it now reads `K>>`. The `K` stands for Keyboard, and it indicates that we are now inside an executing function, but that control has been given over to the keyboard. If you check the Workspace, you see that only the variables that have been defined within the function are listed there. We are really inside the function. However, you can issue anything you like from the keyboard, including things that have nothing to do with the executing function.

As the function has stopped at the breakpoint, you have the opportunity to check and/or change variables as you like. You can also define new ones, but note that these will not be available after you have exited the function again, as the Workspace then shifts to the ordinary one.

12.3.1 Debugging commands to use with breakpoints

After the program has stopped at a breakpoint, there are a few debugging commands available that allows you to stop or continue execution, as well as step through the program line-by-line. All these commands begin with “db”, which is short for “debug mode”.

<code>dbstep</code>	Step through the program line-by-line after having stopped at a breakpoint.
<code>dbcont</code>	Continue the automatic execution of the program either until the next breakpoint or until the program ends.
<code>dbquit</code>	Skip out of debug mode. Note that, for a function this changes the Workspace.

There is also a convenient automatic way to set breakpoints. If Matlab is not able to execute the code, then it is possible to have it go into debug mode instead of ending with an error message.

<code>dbstop if error</code>	Stops the execution of a program if an error occurs, as if a breakpoint had been set at that point. This behavior is permanent until you explicitly take it away.
<code>dbstop if warning</code>	Stops the execution of a program if a warning occurs.
<code>dbclear if error</code>	Deletes the automatic breakpoints at errors.
<code>dbclear if warning</code>	Deletes the automatic breakpoints at warnings.

If we issue `dbstep` from the Command Window, the green arrow skips to the next line, as line 5 is executed, and the contents of line 6 is printed in the Command Window. Issue `fib_nr` from the keyboard to see that the value of the counter variable is 3. If you repeatedly issue `dbstep` a few times, the green arrow will move between lines 6 and 7 as the loop executes, until, eventually, Matlab skips out of the function and back to the ordinary Workspace. Then the output from the function is printed in the Command Window. This way, you can see that the loop is executed three times, for the values 3 to 5 of `fib_nr`.

12.4 Checking programs for correct input

To make a program robust, it is good practice to check whether the input arguments are correct. Particularly if someone else is going to run the program, someone who might not know which conventions you have used as a programmer. Often, what seems simple and straightforward to the programmer does not seem that way to the user. Therefore, input should be checked within the program. If a numerical scalar is expected, for example, you should check that the input is not a matrix or a string. And if it is, you should issue a warning and instructions that the input must be a scalar.

For example, for the Fibonacci program from Section 10.1, we could begin the program by checking if the input is an integer between 2 and 1476. (For higher values, we get overflow which produces `inf`.)

```
if ~(isscalar(fib_max) && isnumeric(fib_max) && fib_max>=2 &&
fib_max<=1476 && fib_max==round(fib_max))
    disp('Input should be an integer between 2 and 1476');

    fib = [];
    return;
end
```

The first line contains five conditions that must all be true for the rest of the code to run. First, we check that the input has only one row and one column; second, we check that it is a number; third and fourth, we check that it is between 2 and 1476; fifth, we check that rounding it to the nearest integer does not change its value (i.e., that it is an integer). Since we are comparing scalars, we use the double-`&` notation described in Section 10.2.

If any of the five conditions is not met, a text warning is issued, the output variable is assigned an empty value, and the function ends; `return` sends it back to from where the function was invoked, either another function or the Command Window. If one does not assign a value to the output variable, Matlab will issue a warning that there seems to be something wrong with the function. To avoid that, we assign it an empty value.

12.5 Use comments

To make future debugging easier, it is also a good practice to use many comments that describe what the program does and how to use it. Also, describe what subsections and complicated individual statements do. If the user, or your future self, has to change the code, such comments are of immense value.

A funny psychological property of most people is that, while working with some code, it seems perfectly transparent and self-explanatory and that no comments are needed. However, when working with code that someone else has written, or even when returning to your own code, it seems opaque and sometimes intentionally written to be misunderstood. The intuition that no comments are needed is usually wrong.

12.6 More on debugging

- There is another debugging feature that causes all commands that are executed in scripts and/or functions to be displayed in the Command Window. See `help echo`.
- There is also a conditional construction of the form `try... catch... end`, where part of the code is executed if the `try`-part runs without errors and the `catch`-part is executed if there are errors. This can be used to make the code able to react to errors while not terminating execution. See `help try`.



Hva får egentlig en ingeniør- eller teknologistudent for 300 kroner?

- Medlemskap i en aktiv studentorganisasjon – hele studietiden
- 150 tillitsvalgte studenter som ivaretar dine interesser
- Jobbsøkerkurs
- Gratis PC-forsikring og gode bank- og forsikringstilbud
- Teknisk Ukeblad og NITO Refleks
- Møteplasser på web 2.0

Flere medlemsfordeler og innmelding: www.nito.no/student

Alle som studerer på ingeniør-, bioingeniør-, sivilingeniør eller andre teknologistudier (høgskolekandidat, bachelor eller master) kan bli medlem i NITO.

NITO NORGES STØRSTE ORGANISASJON FOR INGENIØRER OG TEKNOLOGER



13 Help

There is an abundance of help material for Matlab. In fact, there is so much information out there that one of the main obstacles to finding help is that you might drown in it.

13.1 To find specific information on functions

13.1.1 The `help` command

If you want information about specific functions, this is usually most easily found from the Command Window. If you know the name of the function, then issuing `help functionName` (for example `help abs`) produces a text in the Command Window that describes the function and how to use it.

To be specific, `help functionName` produces the comments at the beginning of the function file. All comments from the beginning until the first non-comment line are reproduced. Therefore, issuing `help randomPlot` produces the comments at the beginning of the script file we developed in Section 8.4.

```
>> help randomPlot
RANDOMPLOT
This script creates a random series of changes and turns them
into an index-like series by multiplying cumulatively.
A series of dates of the same length is also created.
Finally, the series is plotted against the dates.
```

This is one of the reasons that writing good comments is useful.

Help on Matlab's own functions often produces links to both similar commands and to the documentation. Asking for help on, for example, the `abs` function produces the following.

```
>> help abs
ABS Absolute value.
    ABS(X) is the absolute value of the elements of X. When
    X is complex, ABS(X) is the complex modulus (magnitude) of
    the elements of X.
    See also sign, angle, unwrap, hypot.
    Reference page in Help browser
    doc abs
```

The first line of the help text begins with the function name (in caps) followed by a simple description. Then follows a more in-depth description of the function, which, for some commands, can be very long. After this follows a few suggestions that might be alternatives or complements to the function. These are clickable (they are in blue and are underlined), and clicking them produces the help text for the corresponding function. Last, there is a link that opens the Help Browser with the documentation on the function (described in Section 13.2.1).

As an alternative to `help functionName`, you can use `doc functionName`, which immediately opens the Help Browser, similarly to clicking `doc abs` in the help text above.

13.1.2 The `lookfor` command

It is more difficult if you do not know the function name but believe that a certain function does exist. Suppose, for instance, that you have the following problem. When you divide two numbers with each other, the answer is often a fraction. For example, $19/8 = 2.375$. However, another way to represent that answer is to write it as $2\frac{3}{8}$. In this representation, it is made explicit that 19 can hold 2 measures of 8, and that there are 3 eights left. In this case, the number 3 is called the remainder. It is what remains after you have taken out as many full measures of 8 as is possible from 19, in this case. You suspect there is a function for calculating the remainder in Matlab. But how do you find it?

In Matlab, there is a command, `lookfor`, which searches only the first line of the help text where there is usually a brief description of the function. You can often use this to find commands for which you do not know the name. In the example, you could try `lookfor remainder`.

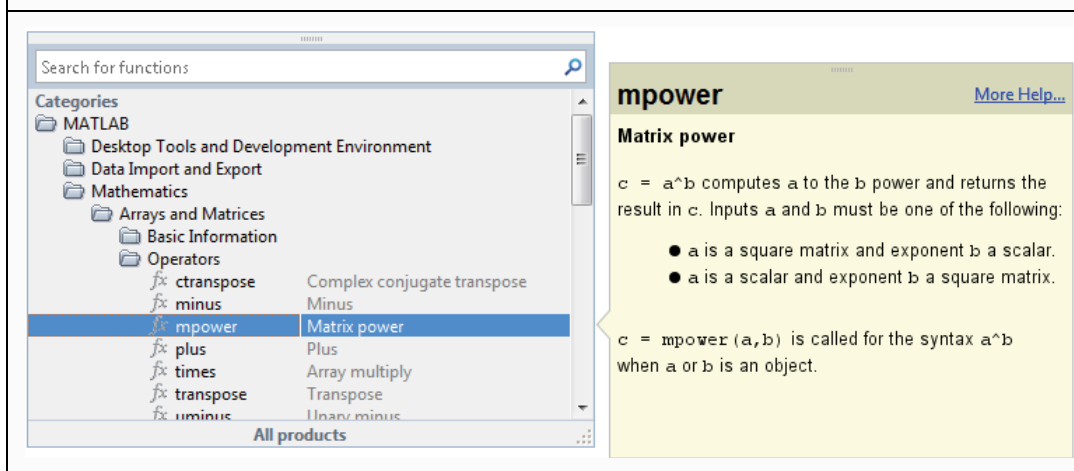
```
>> lookfor remainder
rem          - Remainder after division.
selectionremainder - Remainder stochastic sampling without
replacement.
```

You get two suggestions, and the function you are looking for is the first in the list. Clicking the blue underlined text brings up the help text, where you can check that it is the correct one. Using the function, we get the desired answer.

```
>> rem(19,8)
ans =
    3
```

Note that, the `lookfor`-command sometimes takes a long time to execute.

Figure 13-1: The Function Browser



13.1.2 The Function Browser

Matlab also contains a tool for browsing functions. You open it by choosing **Help > Function Browser**. This opens up the Function Browser (see Figure 13-1) where you can search for functions by category. When you hover over a function name, a balloon-style tooltip appears to the right. Double-clicking the function name copies it to the Editor or to the Command Window, depending on which window is active.



Skatteetaten



Vil du jobbe i et av landets største IT-miljøer?
Vi skal gjøre det kompliserte enkelt

Skatteetaten tilbyr store fagmiljø og utfordrende oppgaver innen:

- > Systemutvikling
- > Service oriented architecture (SOA)
- > Business intelligence (BI)
- > Testledelse
- > Webutvikling
- > IT sikkerhet
- > Infrastruktur
- > Brukergrensesnitt

For nyutdannede IT-spesialister kan vi tilby et to-årig traineeprogram.

For mer informasjon se skatteetaten.no/jobb

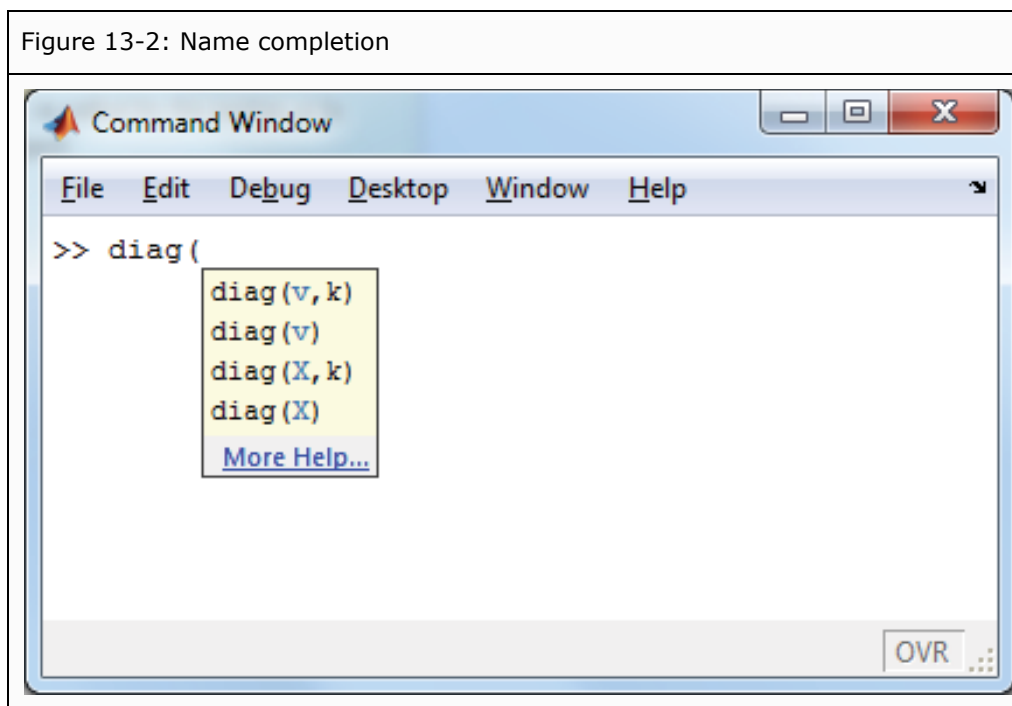
Profesjonell • Nytenkende • Imøtekommende



13.1.4 Function hints

Function hints are enabled by choosing `File > Preferences...`, then first clicking "Keyboard" in the menu to the left, and then checking the tick boxes labeled "Function hints/Enable in Command Window" and "Function hints/Enable in Editor/Debugger", respectively.

When hints are enabled and you type a function name into the Command Window or the Editor, then as soon as the opening parenthesis is entered, a balloon-style tooltip on the function appears with all possible syntax options available for the function.



13.2 To find general information

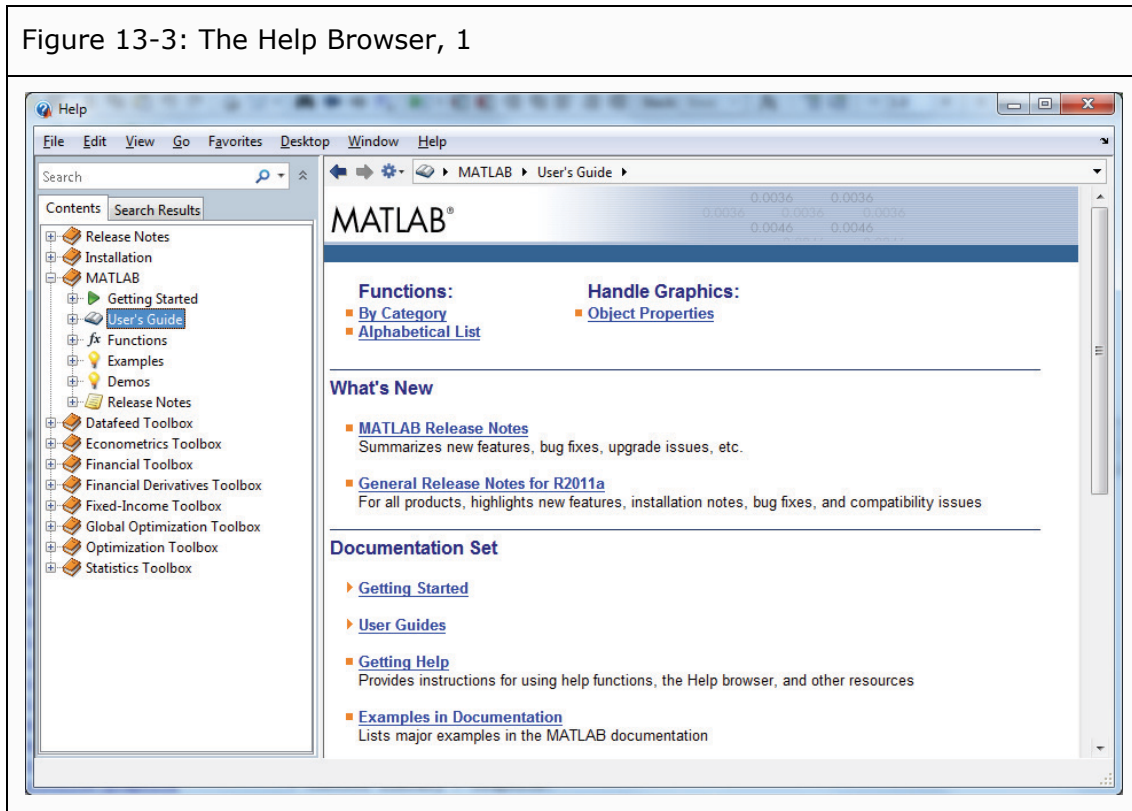
13.2.1 The Help Browser

The Help Browser is used for viewing documentation as well as online help information. You open it by issuing the command `helpbrowser` or by choosing `Desktop > Help`. Figure 13-3 shows the basic layout. (If you do not see the left part of the window, choose `View > Help Navigator` to make it visible.)

The left part of the window is a hierarchical list of topics. At the highest level, the list contains a folder for the main Matlab program and one folder each for installed toolboxes. There is also a folder about installing the program and a folder with release notes. The release notes contain information about changes as compared to previous releases of the program. This can be important if you use old code in new releases, as the way some functions operate may change over time.

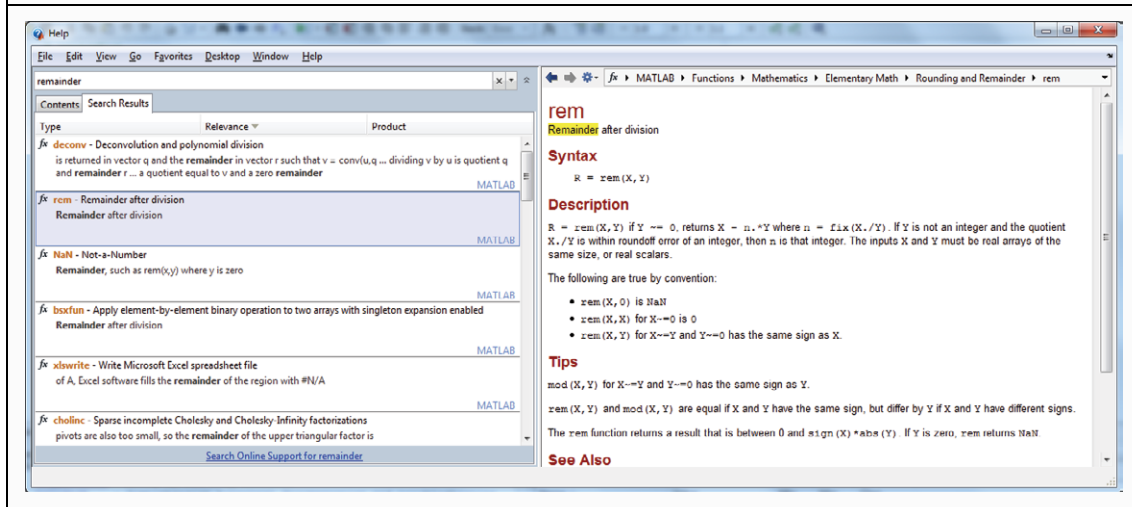
If you expand one of the folders at the top level, a second level is exposed. The main Matlab program and most toolboxes contain a getting started section, a user's guide, a list of functions with explanations, examples, and demos. You can drill down through several levels to find information on specific topics. It is possible to read the user's guides as online books from here.

Figure 13-3: The Help Browser, 1



The documentation is also searchable. As already mentioned, you can find specific information about functions in the Help Browser by issuing the command `doc functionName`, which opens it in the appropriate place.

Figure 13-4: The Help Browser, 2



As an alternative to using `lookfor` (see Section 13.1.2), you can use `docsearch remainder`. This opens the Help Browser and searches for all documents that contain the word “remainder”, which are then listed in the left part of the window. Another way to find the same information is to use the search bar at the top left corner of the Help Browser. In Figure 13-4, we see the Help Browser after having searched for “remainder”.



OLJE- OG ENERGIDEPARTEMENTET



Er du full av energi?

Olje- og energidepartementets hovedoppgave er å tilrettelægge for en samordnet og helhetlig energipolitikk. Vårt overordnede mål er å sikre høy verdiskapning gjennom effektiv og miljøvennlig forvaltning av energiresursene.

Vi vet at den viktigste kilden til læring etter studiene er arbeidssituasjonen. Hos oss får du:

- Innsikt i olje- og energisektoren og dens økende betydning for norsk økonomi
- Utforme fremtidens energipolitikk
- Se det politiske systemet fra innsiden
- Høy kompetanse på et saksfelt, men også et unikt overblikk over den generelle samfunnsutviklingen
- Raskt ansvar for store og utfordrende oppgaver
- Mulighet til å arbeide med internasjonale spørsmål i en næring der Norge er en betydelig aktør

Vi rekrutterer sivil- og samfunnsøkonomer, jurister og samfunnsvitere fra universiteter og høyskoler.

www.regjeringen.no/oed



Se ledige stillinger her

www.jobb.dep.no/oed



13.2.2 Lists of help topics and functions

To produce a list of clickable help topics, issue the command `help` from the Command Window. You may note that the order of the topics is the same as in the search path described in Section 8.3. Part of the list looks like the following.

matlab\general	- General purpose commands.
matlab\ops	- Operators and special characters.
matlab\lang	- Programming language constructs.
matlab\elmat	- Elementary matrices and matrix manipulation.
:	:

For most topics, the first part of the link refers to the name of a folder and the second part refers to the name of a subfolder. For instance, the topics that relate to the main Matlab program are collected in a folder named "matlab" and then general-purpose commands are collected in a subfolder named "general". Toolboxes will have another first part, for example "optim" for Optimization Toolbox. This way, the topics are grouped in an easy to understand fashion. Note that, if you have added your own folders to the search path, these will also be listed here, typically with a text that says that there is no table of contents for these folders.

Clicking a topic produces a list of the commands that belong to the corresponding category. These are in turn clickable and clicking them produces specific help text on the commands. The basic topics include `general` (general purpose commands), `elfun` (elementary math), `specfun` (specialized math), `elmat` (elementary matrices), `graph2d` (two-dimensional graphs), and `matfun` (matrix functions). Note that one of the topics is `helptools`, so clicking that (or issuing `help helptools`) produces a short list of help tools.

13.3 Online documentation from MathWorks

MathWorks (who produces Matlab) also keeps a lot of useful information on their homepage at

<http://www.mathworks.com/support/>

Here you can find the most recent documentation files. Much of the text is, however, the same as in the help files in Matlab. Note also that some information that can be found through the Help Browser is linked to the homepage and that if you have no internet connection, this information cannot be accessed.

On the homepage, there is also a link to Matlab Central, where you can ask questions and exchange files with other users.

13.4 The internet

Searching the internet is often one of the best ways to find help. In many cases, someone else has already had the problem you are trying to solve and has published a solution. A search string with "Matlab" and one or a few keywords will usually bring many hits. Note also that, some researchers that work with Matlab publish code on their homepages for anyone to use.



HELT GRATIS!

S for Skikk & Bank

DU FÅR BOKA HOS DNB

En bok om ting som er greit å vite når du har flyttet hjemmefra.

dnb.no

DNB

Bank fra A til Å




14 Appendix; Commands used in this book

+	csvwrite	helpbrowser
-	cumprod	hist
*	cumsum	hold
/	dbclean	if... else... end
\	dbcont	inf
^	dbquit	inv
%	dbstep	isequal
,	dbstop	ldivide
;	det	legend
:	diag	linspace
'	diff	load
[]	dlmread	log
=	dlmwrite	log10
==	doc	loglog
~=	docsearch	logm
>, >=	echo	lookfor
<, <=	edit	max
&	eig	mean
&&	exp	median
	expm	min
	eye	minus
abs	feather	mldivide
axis	figure	mrdivide
bar	fill	namelengthmax
barh	find	NaN
break	fix	norm
catch	floor	num2str
ceil	fminbnd	ones
chol	for... end	pathtool
clc	format	pi
clear	function_handle	pie
compass	fzero	plot
continue	grid	plottools
csvread	help	plus

polar	semilogx	transpose
polyfit	semilogy	try... catch... end
polyval	sign	uiimport
prod	sort	upper
quad	sortrows	var
quadgk	sqrt	while... end
quadl	sqrtn	xlabel
quiver	stairs	xlim
rand	std	xlsread
randn	stem	xlswrite
rank	str2num	ylabel
rdivide	subplot	ylim
repmat	sum	zeros
return	switch	<Ctrl>-<Break>
rose	times	<Ctrl>-C
round	title	
save	trace	

WANT TO GET TO KNOW BERLIN?


We might have a job for you! Zalando has quickly become Europe's largest online fashion retailer, operating from the heart of Berlin. Want to join our international team?
www.zalando.no/jobb-hos-zalando

 **zalando**



15 Endnotes

1. The notation here works like this: The first word, `Desktop`, refers to the drop-down menus at the top of the window. The `>` means that the next word, `Desktop Layout`, appears only after you have chosen the previous word, etc. So, click on `Desktop` and a menu appears. In that menu, choose `Desktop Layout`, and a second menu appears. In that menu, click `Default`. To choose sub-menus, you do not have to click; it is enough that you point with the mouse on the word.

You may see one or several extra windows as well, depending on whether these were open in the previous Matlab session. For instance, a window labeled `Editor` or a window labeled `Figures` might be open. You can close them by clicking  in the upper-right corner of each window.

Here, as everywhere in this book, the alternative font (such as in `Desktop`) is used for commands and variables in Matlab as well as other tasks that are done on the computer.

2. `>>` is the command prompt in Matlab. When the program has finished all calculations, it types `>>` in the Command Window and waits for the user to issue new commands.
3. You can check this using a command. Issue `namelengthmax`, and Matlab responds with 63.
4. You can change the color-coding by choosing `File > Preferences...`, and then clicking "Colors" in the menu to the left. You will see drop-down menus for choosing colors for a few different cases.
5. You can change it back to a logical variable by issuing a statement like `index_gt5_logic = index_gt5_logic>0`.
6. Here and elsewhere, bold type indicates that a variable is a matrix or a vector.
7. The function name is a word game: *I* and *eye* are pronounced similarly. The reason the function name is not `i()` is that `i` in Matlab is used for the imaginary unit i (i.e., $\sqrt{-1}$).
8. If you are using a program such as Microsoft Word, where it is possible to format the text, you must save the data as "Plain text" (i.e., with the suffix `.txt`).
9. To insert a row in Excel, right-click the line number on the left of the sheet, and choose `Insert` from the context-menu.

10. The data format of the variable containing text data, `labels`, is of a new type, called a cell. Check the Workspace to see that its class is `"cell"`. Cell variables are arrays that can hold other variables of different sizes. The first element of `labels` is a 1x4 matrix of character data, the second is a 1x7 matrix, and the last is a 1x8 matrix. To access, for instance, the second element of `labels`, issue the command `labels{1, 3}`, where you enclose the row and column numbers within curly brackets. To read more about cell variables, see the Matlab documentation.
11. You can also open a figure window by choosing `Desktop > Figures` from the drop-down menus.
12. There are, of course, also commands for time operations, although they tend to be complicated to use with graphs. For examples of time operations, see `help datestr` and `help datenum`.
13. If you open the new window by first selecting the Editor (clicking inside it) and then choosing `File > New > Function`, a new window will open with some text, such as the code word `function`, already filled in.
14. As mentioned in Section 9.2, the function we want to solve can also be entered as an anonymous function, which, in this case, would be faster.

Actually, it can even be entered as a string, as in `x_zero=fzero('sin(x) + exp(-x)', 3)`.



"I studied English for 16 years but...
...I finally learned to speak it in just six lessons"

Jane, Chinese architect

ENGLISH OUT THERE

Click to hear me talking before and after my unique course download

